# An Implementation of Process Swapping in MINIX (A Message Passing Oriented Operating System)

by

Stanley George Kobylanski

B.S., Pennsylvania State University, 1972

---

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

Approved by:

Masaaki Mizuno

Major Professor

CONTENTS

LIST OF FIGURES

LIST OF TABLES

## ACKNOWLEDGEMENTS

My wife, Janine, and my son, Abraham, have given me more love and support within the past year than any man deserves. It is impossible to return so much, but I will try.

I would like to thank Dr. Virgil Wallentine and Dr. Maarten Van Swaay for serving on my committee and for exposing, to me, a small portion of their wealth of knowledge and wisdom.

I would also like to thank my advisor, Dr. Masaaki Mizuno, for sharing his theoretical knowledge, his technical expertise, and his humor. His guidance has kept me within the bounds of the project while allowing me the latitude to explore until I could reach and defend my own conclusions.

Finally, I would like to thank Charles Clouse, AT&T Document Development Organization, for his help in formatting this document.

# CHAPTER 1

## 1. INTRODUCTION

### 1.1 Purpose of Operating Systems

A computer system consists of hardware and software that cooperate to do useful work. Computer hardware consists of the physical devices that you can see and touch, such as memory circuits, disk drives, tape drives, terminals, printers, light pens, keyboards, etc. They provide most of the key computer resources: storage for data, computing power for processing data, and devices for input and output of data. Another key resource is the data itself.

Computer software, often referred to as computer programs, consists of a sequence of logical instructions that the computer hardware performs to achieve a desired result. Software and hardware, teamed to form a computer system, have the ability to do various functions, from check book balancing to highly complex programs that can mimic some parts of human behavior.

An operating system is a computer program that:

- manages the resources of a computer, and
- provides easy access to complex hardware.

An operating system allows expensive hardware resources to be shared by many users. For

example, a laser printer that provides high quality printing might be too expensive for a single user, but can be affordable when the cost is shared by many users. Hardware resources can also be hard to use because of their complex interfaces. An operating system hides the hardware complexity by converting user requests so that the hardware can accept them.

Hardware provides "raw computing power" and operating systems make this power conveniently available to users.

### 1.2 MINIX Operating System

**1.2.1 Introduction** There are many different operating systems, each built to satisfy certain requirements. MINIX is a general purpose, multiprocessing, multiuser, operating system designed to serve as an aid in teaching operating system concepts. To that end, it was designed to:

- be small, so that it is not overwhelming and can be understood by a student.
- provide an outward appearance (user interface) that mimics a popular operating system called version 7 (v7) UNIX[1] (hence it's name *MIni uNIX*). MINIX includes most of the system calls (basic operating system commands), features, and supporting programs provided by UNIX.

1. UNIX is a trademark of AT&T.

- be modular to aid in comprehension and to encourage modification.
- run primarily on the IBM-PC and most compatibles but it has been ported to other machines (e.g., ATARI-ST).
- support multiple users. However, the processing power of the IBM-PC (Intel 8088) is limited such that only 1 user can comfortably be supported. On processors more powerful than the IBM-PC, such as the IBM-AT (Intel 80286) or Intel 80386, more than one user can be supported.

### 1.2.2 MINIX and UNIX

**1.2.2.1 General Description** A brief description of MINIX and UNIX is provided here, for more information see references [1] and [2]. MINIX and UNIX consist of four major components: process control, memory management, file system, and input/output.

- Process control handles process creation, communication, scheduling, termination, and miscellaneous process services (suspension, resumption, memory growth, etc.). In MINIX and UNIX, this function is provided by the kernel.
- Memory management allocates and deallocates main memory as needed by processes. In systems that provide swapping or paging, it manages the transfer of process images to and from secondary memory as well as allocation and deallocation of secondary memory. In MINIX, this function is provided by the memory manager (MM). The UNIX kernel provides this function.

- File system manages secondary memory for efficient storage and retrieval of user data. In MINIX, this function is provided by the file system (FS). The UNIX kernel provides this function.

- Input and output procedures provide controlled access to peripheral devices such as terminals, tape drives, disk drives, and network devices for user processes. In MINIX and UNIX, these functions are provided by device drivers (also known as tasks in MINIX) that are linked with the kernel.

**1.2.2.2 UNIX Description** Unix was designed in the early 1970's to support general computer science research and to provide a custom work environment for its creators. It has evolved from a custom work environment to a popular general purpose operating system.

UNIX is an example of a layered operating system. Its high level architecture is shown in Figure 1-1. The rings in the figure represent levels of interaction and privilege. A program in a ring can only interact with or get services from programs in adjacent rings. For example, the cc program cannot directly interact with (or request services from) the kernel.

Only the operating system (kernel) interacts directly with the hardware, providing common services to programs and insulating them from the hardware idiosyncrasies. Programs interact with the kernel by invoking a well defined set of system calls. An advantage of this design is that the rings can be extended as much as the user prefers.

Within the kernel, there is no structure nor information hiding. It consists of a collection of

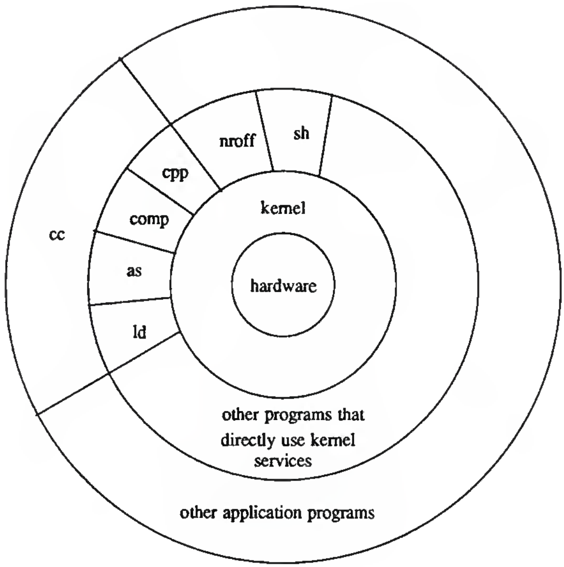


Figure 1-1. Architecture of UNIX Operating System

procedures that are compiled into a single object file. System calls from user processes execute a special instruction called a trap. This instruction switches the machine from user mode to kernel (or supervisor) mode. Control is transferred to the kernel which acts on behalf of the user process. The kernel is not a separate set of processes that run in parallel with user processes, but it is a part of each user process. In effect, the user process becomes the kernel to perform the protected operating system functions in the kernel mode. When the system call completes, the machine is switched back to user mode and the

process resumes in user mode.

As an example, consider the *open* system call which prepares the user process to access a file for reading and/or writing. The user process issues the system call, in the C programming language, as:

```
fd = open(path,mode);
```

where *fd* is the file descriptor used for future file access, *path* is the path and name of the file to be opened, and *mode* is the access permission: read, write, or both. The *open* system call has an entry point in the system call library. The library, encoded in assembly language, contains special trap instructions, which when executed cause an "interrupt" that results in a hardware switch from user mode to kernel mode. For each user process in UNIX, there exists a user stack for use in user mode and a kernel stack for use in kernel mode. The switch to kernel mode causes a switch to the kernel stack and allows the user process to execute the kernel procedures for opening a file. Without getting into great detail about the UNIX file system, the file, if found, is checked for access permissions for this user, prepared for access, and the file descriptor is returned to the system call library *open* routine. The system call library executes another trap instruction that results in a hardware switch back to the user mode. The file descriptor is returned to the user process which resumes in user mode.

**1.2.2.3 MINIX Description** MINIX is an example of a client-server system. Although it provides the version 7 UNIX user interface, the internal structure and operation are different

than UNIX. In a client-server (or message passing) model, such as MINIX, as much functionality as possible is removed from the operating system leaving a minimal kernel. Most of the operating system functions are contained in user processes. To request a service (system call), a user process (known as a client process), sends the request to a server process which does the work and returns the results. The kernel handles the communication between the clients and servers. The servers run as user mode processes and do not have access to the hardware. The structure of MINIX is shown in Figure 1-2.

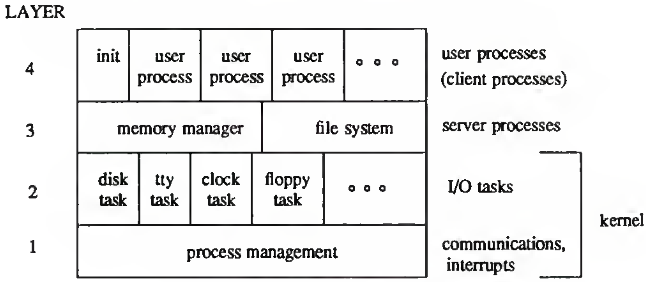


**Figure 1-2.** Architecture of MINIX Operating System

The layers are similar to the UNIX architecture in that a program can only interact with adjacent layers.

Layers 1 and 2 are compiled into 1 program called the kernel. The process management layer handles all interrupts and traps, and provides message communication between all processes. Layer 2 consists of input/output processes, typically called tasks or device

drivers. Although linked together, they run as separate processes. They provide the device dependent services (physical I/O with specific hardware devices).

Layer 3 contains two processes that support all system calls for user processes. Layer 3 processes provide the device independent services (logical I/O common to all devices).

Layer 4 contains all user processes. System calls from user processes are handled as a client-server transaction. A user process requests a system service by SENDING the request to the appropriate server (FS or MM) and waiting for the results. The server RECEIVES the request, performs the service (which might involve communications with other servers) and returns the results. Then the user process resumes.

For comparison with UNIX operation, a MINIX *open* system call is described. A user process executes the *open* system call using the same syntax and interface as UNIX. The system call library function converts the call to SEND and RECEIVE statements. These are MINIX communication primitives. The library function then executes a trap instruction that causes a software interrupt that is serviced by the kernel. The MINIX kernel puts the SEND and RECEIVE requests on an internal message queue for FS. The user process (via the library function) is then blocked until both requests complete. When FS issues a RECEIVE request to accept new work, the MINIX kernel delivers the user process message to FS. FS copies the message to its data region, determines that it is an *open* system call, and begins to perform the function. All data relating to the file system is maintained within FS. It is not directly accessible by any other process, including the kernel. When FS completes the function, the results are returned using the SEND request with the user

process as the destination. The user process (via the library function) blocked on RECEIVE, accepts the results immediately, freeing the FS SEND block. FS then issues a RECEIVE request for ANYone and awaits new work. When the user process is scheduled to use the CPU, it proceeds from the reception of the system call results.

Note the difference in system call handling between UNIX and MINIX. In UNIX, the user process performs the system call within the boundaries of the kernel procedure. In MINIX, system calls are performed by independent server processes for the waiting user process (client). This has implications on the design presented later in this document.

An empirical study of message oriented operating systems versus procedure oriented operating systems is provided in reference [3]. MINIX fits the definition of a message oriented system and UNIX somewhat fits the definition of a procedure oriented system. The study demonstrates that those two types of operating systems are duals of each other and that a system constructed according to one model has a direct counterpart in the other. It concludes that neither model is inherently preferable, the main consideration being the nature of the machine architecture, not the application that the system will support.

**1.2.3 MINIX Limitations** MINIX and its host machine(s) do have some limitations. Some of the notable limitations are:

- Program Size. MINIX programs are limited to 64K bytes if compiled as non-separate text and data and 128K bytes if compiled as separate text and data. In the latter case, the text and data regions each have a limitation of 64K bytes. The program size

limitation is due to the hardware memory management unit of the host computer.

- Memory Size. The main memory size for a MINIX system is limited to 640K bytes. Again, this is a hardware imposed limitation.

**1.2.3.1 MINIX Problem** A user session with MINIX occasionally results in frustration because of the main memory size limitation. Main memory use on an IBM-PC running MINIX varies, but a typical distribution is:

| MEMORY LOCATION | RESIDENT PROGRAMS/DATA |
|---:|---|
| 0 - 120K | MINIX Operating System |
| 120K - 360K | root File System RAM Disk |
| 360K - 640K | User Process Area |

The amount of main memory available for user processes is about 280K bytes. This allows about four 64K bytes processes to run concurrently. Additional processes will be rejected due to lack of main memory, even though the currently running processes might be blocked and ample CPU power is available. On a multi-process system such as MINIX, it is not uncommon to have many processes active, even for a single user. For example, the user might type:

```
$ ls -al | grep "[a-m]*.c" | more
```

to list specific files in the current directory and view them a screenful at a time. The count of active processes is: user command shell, a shell spun off to handle the multiple

commands line, ls command, grep command, and the more command, for a total of 5 processes. In this example, the user command shell is blocked until all commands complete; it can be swapped out if necessary. The design and implementation discussed in this paper address the memory limitation problem.

# CHAPTER 2

## 2. REQUIREMENTS

This chapter presents the goals and objectives that the design and implementation must meet to provide a useful enhancement to the MINIX operating system.

### 2.1 Extend Memory

The 1.3 version of MINIX provides about 280K bytes of main memory (about 520K bytes on an AT model) for user programs. Occasionally, this limit is reached without exceeding the processing capability of the CPU. This results in a refusal to perform the user requested command and causes user disappointment and frustration. The system should provide a means to overcome the main memory limitation and allow full use of the CPU. This will allow more useful work to be done and will promote user satisfaction.

### 2.2 Maintain Existing Functions

Operating system modifications should not affect the current functions provided by MINIX. Additional functions are permissible only if they do not affect the current functions.

### 2.3 Maintain MINIX Structure

As described in Chapter 1, MINIX is a highly structured, modular operating system. The design should maintain that structure to allow future enhancements to be developed with the same philosophy.

### 2.4 User Administration

The installation and maintenance of the modification should be minimal and straightforward. Automated installation procedures should be provided. Maintenance procedures should be clearly documented.

### 2.5 Performance

The implementation should incur minimal performance degradation and main memory usage so as not to offset the performance gain. When the new functionality is not in use, the system performance should equal that of the current system. When in light use, the system performance degradation should not exceed 25%. When in heavy use, the CPU limitations might be exceeded; this cannot be avoided.

# CHAPTER 3

## 3. DETAILED DESIGN

### 3.1 Introduction

This chapter presents and defends a design that meets the requirements presented in Chapter 2. A process swapper that shuttles process images between main memory and secondary memory has been chosen to satisfy the requirements. The design is applied to thc 1.3 version of MINIX.

### 3.2 Process Swapping

**3.2.1 Purpose of Swapping** Process memory images[2] must reside in main memory to run. If main memory is not available, a process cannot be created. Process swapping allows more processes to run than can fit into the available main memory. It does this, transparently to the user, by shuttling process images between main memory and secondary memory (called the swap device). Clearly, a process on secondary memory requires longer access time than a process in main memory, so low priority, inactive processes are chosen for temporary swap out. When the swapped out processes are ready to run and main memory is available, they are moved back into main memory.

---

2. A process memory image consists of the data and instructions that must reside in main memory for the process to run. It can include the process text, data, stack, and the process table slot. Within the context of this document, the process memory image consists of the process text, data, and stack regions which can be swapped out and in.

**3.2.2 Swapping Functions** The functions provided by swapping are:

- provide for process creation when main memory is not available by creating the new process image on the swap device.
- swap in runnable processes and, if necessary, free main memory by swapping out inactive processes.
- choose appropriate processes for swap in/out.
- manage space on the swap device.

### 3.2.3 Alternatives to Swapping

**3.2.3.1 Increased Memory** It is not feasible to increase the amount of main memory beyond 640K bytes on an IBM-PC. The IBM-AT model can provide up to 384K bytes of extended memory. Currently, MINIX can use the extended memory only to support the root file system RAM disk. This frees 240K bytes of main memory for user programs and is an excellent alternative to swapping for that machine.

**3.2.3.2 Demand Paging** Swapping transfers the entire process image between main memory and the swap device. An alternative to swapping is demand paging which transfers individual memory pages instead of entire processes to and from a secondary device. Demand paging permits greater flexibility in mapping the virtual address space of a process into the physical memory of a machine, usually allowing the size of a process to be greater than the amount of available physical memory and allowing more processes to fit simultaneously in main memory. Unfortunately, the IBM-PC does not provide hardware to

support demand paging.

### 3.3 General Design

**3.3.1 Introduction** The swapping design affects 3 parts of the MINIX operating system: the kernel, memory manager (MM), and the file system (FS). An overview of MINIX with the swapping design is shown in Figure 3-1. The figure shows most MINIX components but shows the communications only concerned with swapping.

Recall from chapter 1 that tasks are usually interfaces to hardware devices. They are all linked together in the MINIX kernel, but they each behave as a separate process. This gives them independence and access to kernel variables, including the kernel process table. A new task, the swap task, was added to the kernel to manage the overall swapping function. Unlike other tasks, it performs no hardware interfacing. It communicates with other tasks and the memory manager (MM), using normal MINIX communications primitives, to manage the swapping function.

The swap task relies on the memory management functions of MM. MM is responsible for setup and completion of all swap ins and swap outs. MM initiates fork and exec swaps and, upon command from the swap task, performs swap ins and forced swap outs. MM requests physical I/O for swapping via normal user system calls to the file system (FS).

The physical swapping input/output and swap device management is performed by the file system (FS). The swap device is the area on secondary memory used to store process memory images that have been swapped out of main memory.

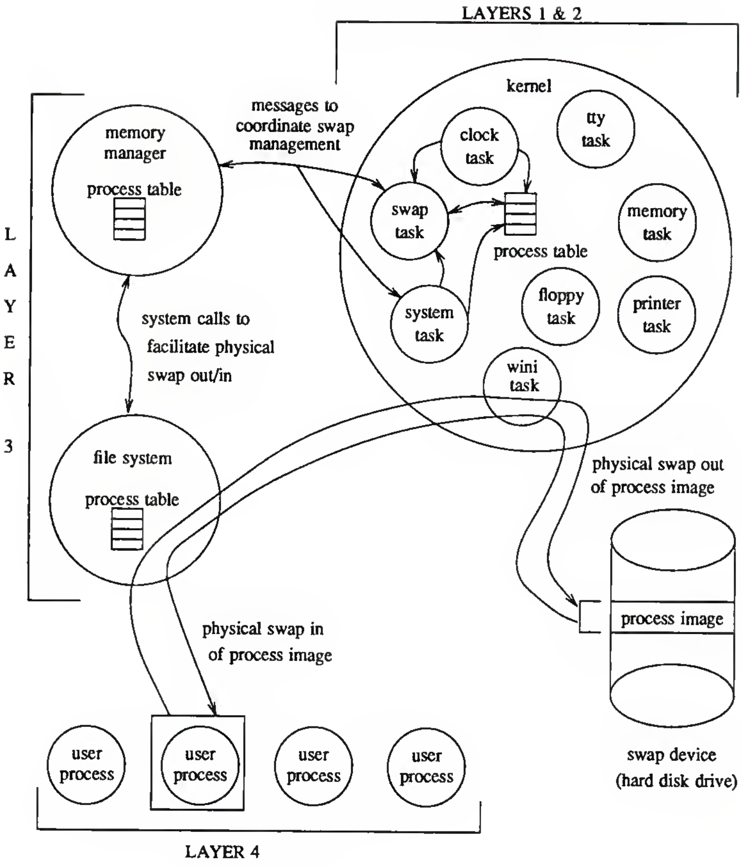


**Figure 3-1.** MINIX Process Swapping Overview

**3.3.2 Overview** This part describes a typical situation that involves the entire swapping design (Figure 3-1).

Assume that a user process issues the *fork* system call to create a new process that is a copy of itself (i.e., create a child process). This is typical in multiprocessing type operating systems. The memory manager (MM) receives the request from the user process and starts the fork function. *fork* must allocate main memory for the new process image. If main memory is not available, then the new process image is not copied to main memory but is copied to the swap device instead. This is done within MM using normal user system calls to FS. The calling user process memory image is written to a specific directory within a file system on secondary memory (disk). The directory, called the swap device, is a repository for all swapped out process images. When that is complete, MM sends a message (via the system task) to the swap task to indicate that a runnable process is on the swap device. After the swap out MM returns the process id of the child process to the parent process just as in a normal *fork*.

The swap task receives the message and chooses the most eligible, runnable process on the swap device to swap in. It sends a swap in request to MM and awaits the results.

If main memory is available, MM swaps in the process using normal user system calls to FS to read the user process memory image from the directory (swap device) and copy it to main memory. When the swap in has completed, MM notifies the swap task that the process image has been swapped in. If main memory is not available, MM notifies the swap task of the failure. MM also provides information to the swap task to aid in choosing

a process to swap out.

If the swap in succeeded, the swap task checks for more runnable processes on the swap device and attempts to swap them in in the same manner. If the swap in failed, then to free main memory for the runnable process on the swap device, the swap task chooses an eligible process in main memory to swap out. It uses the information provided by MM to aid in selection. If it cannot find a process to swap out, the swap task idles until it receives an indication that main memory has been freed or that a process in main memory might now be eligible to swap out. When a process is chosen for swap out, the swap task sends a swap out request to MM and awaits the results.

MM swaps out the requested process image using normal user system calls to FS to write the user process memory image to the swap device. When the swap out has completed, MM notifies the swap task.

The swap task then chooses the most eligible runnable process on the swap device to swap in. As previously described, it requests MM to swap in the process.

This cycle continues until there are no runnable processes on the swap device. Processes chosen by the swap task for swap out can be runnable or blocked on a system call. Runnable processes are eligible for swap in immediately. Blocked processes are not eligible for swap in until the system call has been completed. When a blocked process on the swap device becomes runnable, a message is sent to the swap task to start the swap in procedure.

**3.3.3 Swapping Functions** From the Overview, three main swap functions are identified: managing space on the swap device, swapping processes out of main memory, and swapping processes into main memory.

**3.3.3.1 Swap Device** Swapping introduces another resource that the operating system must manage, the swap device. The swap device is the area on secondary memory used to store process memory images that have been swapped out of main memory. The swap device can be considered an extension of main memory with a large size and limited capability. Its capability is limited to storing process memory images because processes cannot use the CPU while on the swap device. Its advantages are that a process can be kept alive while on the swap device, a system call placed before the process was swapped out can progress (because it is performed by another process: MM or FS), and the main memory that the process is not using can be used by another running process.

**3.3.3.2 Swapping Out Processes** Figure 3-2 shows the original MINIX user process states. Processes created by *fork* and overlaying programs introduced by *exec* ("new proc") are placed in main memory and then the process is placed on the RUN queue ("on RUN queue"). However, if main memory is not available, then the process creation fails. With the introduction of swapping, these failures become candidates for swap out. Figure 3-3 shows a new state ("SWAPPED & RUNNABLE") to which these failures go when main memory is not available. In this state the process memory images are on the swap device. Later, when main memory is made available, they are swapped in (see "Swapping In Processes").

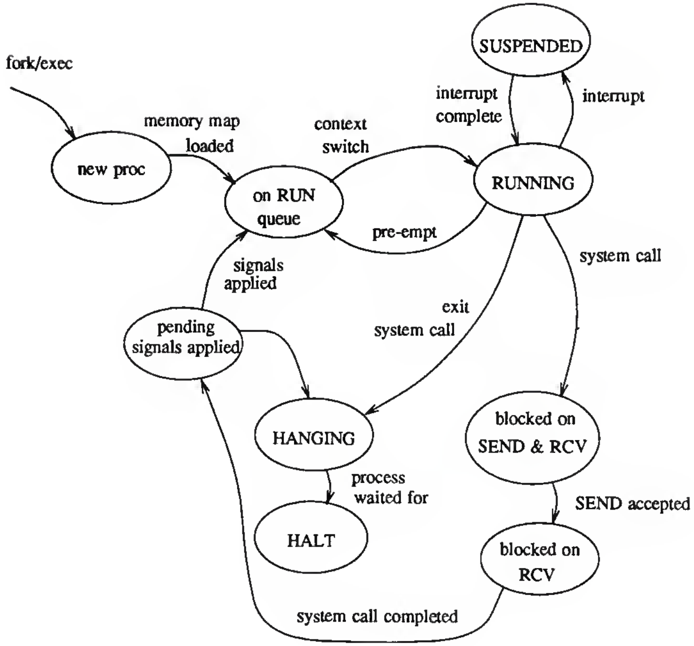


**Figure 3-2.** Original MINIX User Process States

It is necessary, at times, to force a process out of main memory to free space for a runnable process currently on the swap device. Processes must be chosen carefully for forced swap out. Recall that when a user process performs a system call it is blocked until the system call is completed by MM or FS. Most system calls result in data being transferred to or from the user process data region in main memory. MM and FS rely on the fact that a

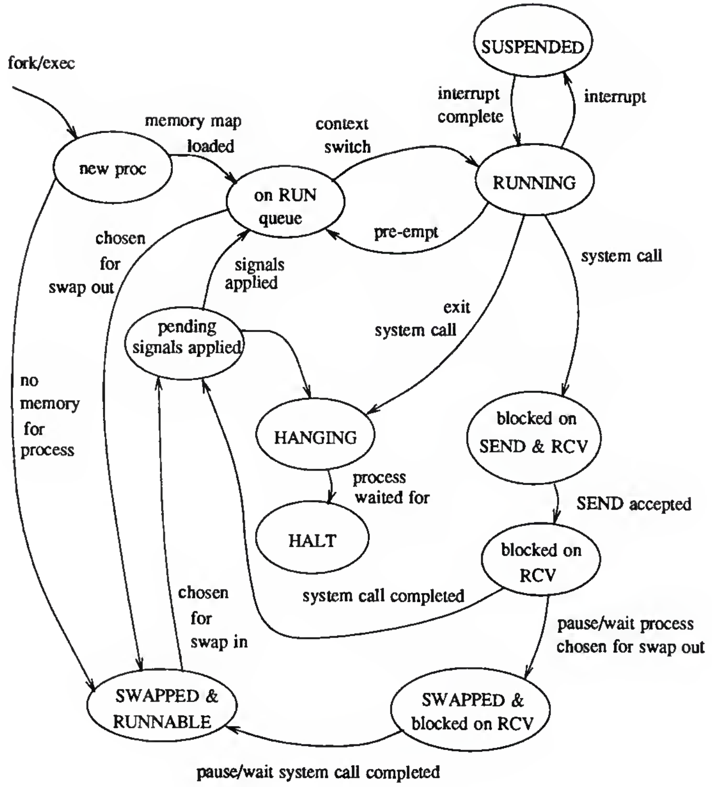


**Figure 3-3.** MINIX User Process States with Swapping

process will always be in main memory. Therefore, a process that is chosen for swap out should not be waiting for I/O. This affects which processes are eligible to swap out.

For example, suppose that a user process wishes to write some data to an I/O device (state "RUNNING" in Figure 3-2). It issues a *write* system call to the operating system. This is done by SENDing a message to FS and awaiting the result by issuing a RECEIVE from FS. The user process is therefore blocked on SEND and RECEIVE ("blocked on SEND & RCV"). When FS issues a RECEIVE (to accept new work from ANY source), the message passing mechanism of the operating system passes the system call *write* request to FS. The user process is now freed from the SEND block and is blocked only on RECEIVE from FS ("blocked on RCV"). At this point FS is dedicated to processing the user's *write* request. There is no way to stop it. Data will be read from the user's area and copied to the I/O device. If the user process is swapped out BEFORE the data has been retrieved from its data region, FS must wait until the user process is swapped back into core. Currently, FS is not designed to wait for an indeterminate amount of time for that to occur. Similar problems exist with other system calls.

In UNIX, the problem is simplified because the user process itself traps to kernel mode to perform operating system functions. When the user process is in core and in the kernel mode, just before it does an I/O operation, it marks itself as being ineligible for a swap out. It stays marked until the I/O has completed. When the user process is swapped out, no progress can be made on the system call because the user process itself performs the protected kernel procedures in kernel mode. Therefore, an I/O operation cannot begin while the process is swapped out.

Clearly, the UNIX solution cannot be used. In MINIX, one solution is to mark all user

processes that do system calls requiring their data region to be read or written as ineligible for swap out. This requires marking nearly all system calls because the message passing scheme uses memory pointers to the user area to pass system call parameters and to return data and responses. It would not make a significant difference to single out the few that do not access the user process data region.

Another solution is to buffer I/O requests for swapped out processes so that I/O is delayed until the process is swapped in. This is a complex modification to MINIX and the buffering required would increase the operating system size.

This design chooses, for forced swap out, those processes that are blocked for an indefinite amount of time and are expecting a minimal amount of I/O. Conveniently, MINIX clearly identifies such processes in the MM process table. They are processes blocked on the system calls *pause* and *wait*. They are ineligible to run until the system call completes so they are excellent candidates for swap out. This is illustrated in the user process state diagram Figure 3-3 by the new state ("SWAPPED & BLOCKED ON RCV"). These two system calls generate minimal I/O that is easily buffered by the kernel until the process is swapped in. When the system call completes, the ("SWAPPED & RUNNABLE") state is entered and the process is ready for swap in.

A secondary choice for processes to swap out, is processes that are not blocked at all. They are expecting no I/O because they are currently CPU bound. It is an easy matter to swap them out ("SWAPPED & RUNNABLE"). They are not an ideal choice for swap out because they are runnable, but they should rarely get chosen. In situations where they do

get swapped out (no blocked processes remaining in core), they provide a method for sharing the CPU fairly among all runnable processes.

A third possibility for forced swap out is processes blocked on both SEND and RECEIVE. This has been considered and it is feasible, but because of the added complexity, it was not implemented.

Another problem with swapping out processes concerns the location in main memory of the swapped out process and how it relates to the process to be swapped in. MINIX uses the first fit algorithm when allocating main memory for new processes. Suppose that a number of processes reside in main memory with a total free space of 20K bytes. A process on the swap device with a size of 18K bytes is runnable and should be swapped in. However, the free main memory is fragmented and the largest amount of contiguous free main memory is 10K bytes (Figure 3-4a). Should this require a swap out of a program or group of programs whose size is >= 18K bytes, or a swap out of a program(s) next to a free block(s) such that program(s) size + free block(s) size >= 18K bytes? This idea is suggested in references [4] and [5]. One advantage is that no processes are swapped out if a large enough cluster cannot be found. This reduces the number of unnecessary swap outs. (The standard UNIX swapper swaps out processes even though doing so might not result in a large enough area for the incoming process. So unnecessary swapping occurs.)

Including this type of requirement in the algorithm for choosing a process for swap out makes it more complex and results in a smaller set of processes from which to choose. Recall that only a special subset of all user processes is eligible for swap out (runnable or

blocked on *pause* and *wait*). An alternative is to compact memory before the swap in such that a contiguous block of free memory of size 20K bytes resides in high memory (Figure 3-4b). The process can then be swapped into the free memory (using first fit) and a potential swap out is eliminated (Figure 3-4c). The performance cost of memory compaction on the IBM-PC is certainly less than the cost of a swap out to secondary memory. In addition, compaction simplifies the algorithm to choose a process to swap out. For example, if an 18K bytes runnable process is on the swap device and total free main memory is 10K bytes and is fragmented, then all that is required to provide space for the swap in is to find an eligible incore program >= 8K bytes, swap it out, and, if necessary, compress memory. The process can then be swapped into the free memory. The overhead of a compaction is incurred but is justified if the following is considered. The amount of main memory available for MINIX user processes is small (about 280K bytes) and the actual number of processes is small (process table can accommodate 15 user processes maximum). Because there is not a lot of main memory to compact, compaction is quick and because there are not a lot of processes from which to choose, the compression algorithm will find an eligible process for swap out much more often than the other algorithm. In addition, the compression algorithm can choose more appropriate processes for swap out.

**3.3.3.3 Swapping In Processes** It now becomes an easy matter to determine which processes are candidates for swap in: processes in the process state ("SWAPPED & RUNNABLE"). A priority, based on residence time, is placed on the processes in that state to facilitate the choice.

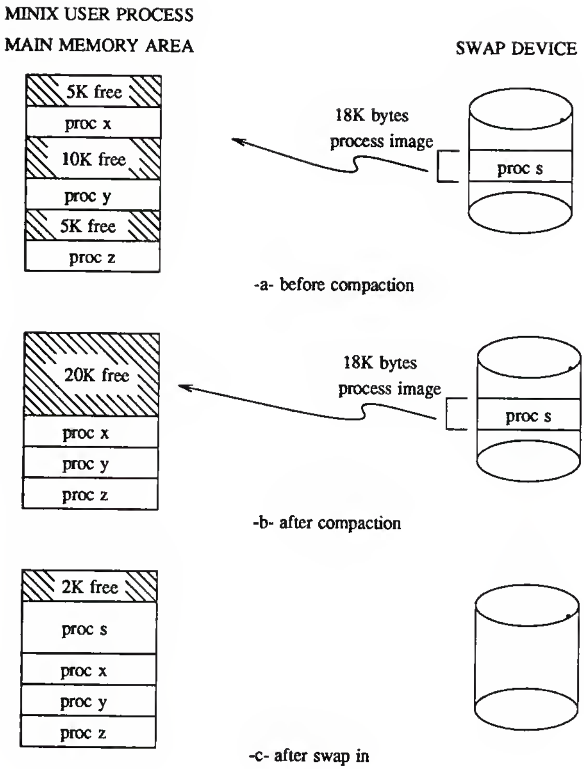


**Figure 3-4.** Illustration of Compaction

### 3.4 Detailed Module Description

**3.4.1 Introduction** The 3 swapping functions just described are provided by various parts of the operating system: kernel provides management and decision making, MM provides coordination, and FS provides primitive swap out/in operations. The design changes to the kernel, MM, and FS to provide swapping are described here.

### 3.4.2 Kernel - Management and Decisions

**3.4.2.1 Swap Task** The swap task makes all swapping decisions. The swap task was implemented as a kernel task for the following reasons:

- it can be closely tied to process scheduling
- it needs timely access to some kernel variables.

The swap task, normally idle, awaits messages concerning system swap status and it responds appropriately. Two variables define the state of the swap task: swap_state and inprogress. Swap_state can have the following values:

- IDLE - no runnable processes on the swap device
- SWAP_IN - runnable process on the swap device.

A runnable process is a process that is not blocked on SEND and/or RECEIVE. Inprogress has the following values:

- NONE - no swap operation in progress
- SWAP_IN - a swap in is in progress

• SWAP_OUT - a swap out is in progress.

A complete swap task state diagram is shown in Figure 3-5.

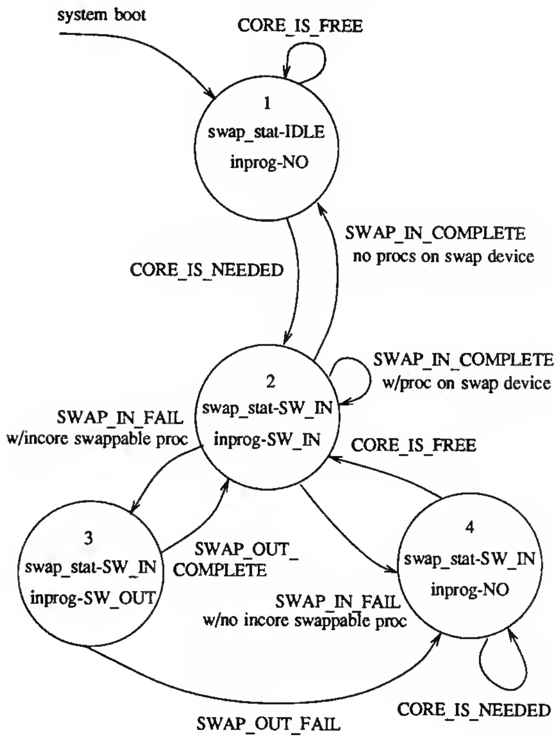


**Figure 3-5.** State Diagram for Swap Task

The swap task is described in terms of its state and transitions.

**3.4.2.1.1 Swap Task States** State 1 is entered at system startup. There are no runnable processes on the swap device and the swap task idles whenever in this state.

State 2 is entered when a runnable process is on the swap device and an attempt is being made by the swap task to swap it in.

State 3 is entered when a runnable process is on the swap device, main memory is not available, and a swap out is being attempted by the swap task to free main memory.

State 4 is entered when a runnable process is on the swap device, main memory is not available, and no in core processes are eligible for swap out. The swap task idles until memory is freed or a process becomes eligible for swap out.

**3.4.2.1.2 Swap Task Transitions** The swap task can receive six different messages from MM and other tasks. The following describes how each of the messages (or events) affects the swap task states and how the swap task responds.

CORE_IS_NEEDED: This message is received when a runnable process is on the swap device. It is the swap task's responsibility to move it into main memory as quickly as possible. This message is sent from:

— MM when a process, on the swap device, that was blocked on the *pause* or *wait* system call has become unblocked.

— MM (via the system task) when a *fork* or *exec* swap has occurred.

If the swap task is in state 1, the swap task's response is to change the swap status from

SWAP_IDLE to SWAP_IN (state 1 to state 2) and try to swap in the runnable process. All runnable processes on the swap device will be evaluated and the most eligible will be chosen (see "Swap In Algorithm" below). The swap task then sends a SWAP_IN_REQUEST message to MM to swap in the chosen process. MM will attempt to swap in the process and will return either a SWAP_IN_COMPLETE or SWAP_IN FAIL message to the swap task. The swap task can have, at most, one swap in or one swap out operation in progress.

SWAP_IN_COMPLETE: This message is received when a swap in, requested by the swap task (from state 2), has successfully completed. It is sent by MM after swap in completion and should be received only while in state 2. The swap task responds by readying the process for execution. Then the swap task checks for more runnable processes on the swap device (and stays in state 2 or goes to state 1).

SWAP_IN_FAIL: This message is received when a swap in, requested by the swap task (from state 2), has failed. It is sent by MM after attempting a swap in. The reason for failure is assumed to be not enough main memory for the process. MM also sends additional information to help the swap task choose a process for swap out: a list of all processes that are blocked on the *pause* and *wait* system calls and the amount of main memory required to swap in the requested process. Based on the information returned by MM, the swap task responds by choosing a process to swap out so that main memory can be freed to allow the runnable process into main memory. It then marks the chosen process as SWAPPED and sends a SWAP_OUT_REQUEST to MM (state 3). If an eligible process

for swap out cannot be found (state 4), then the swap task sends a NO_SWAP_OUT message to the waiting MM and idles until a CORE_IS_FREE message is received.

CORE_IS_FREE: This message is received when a parameter that controls swap out has changed. If the swap task is in state 4, it can re-evaluate all in core processes to find an eligible swap out candidate. Changeable swapping parameters are: amount of main memory available and process residence time. The other parameter is process memory image size. This message is sent from:

— MM (via the system task) when a process has exited meaning that main memory has become available.

— MM when a process has blocked itself (i.e., it has called the *pause* or *wait* system call), meaning that main memory can be freed by swapping out the blocked process.

— Clock task after the process residence times have increased. This means that a process might now be resident long enough to be eligible for swap out to free main memory.

The swap task (state 4) responds by evaluating all runnable processes on the swap device and trying to swap in the most eligible, hoping that enough core is available (state 2). If the swap in fails, then the swap task will evaluate all processes in main memory for swap out (and go to state 3 or 4).

SWAP_OUT_COMPLETE: This message is received when a swap out, requested by the swap task (from state 3), has successfully completed. It is sent by MM after swap out completion. The *fork* and *exec* swaps are reported by the CORE_IS_NEEDED message. A

forced swap out is performed only when main memory space is needed for an incoming process. So the reception of this message means that there must be a runnable process on the swap device. The swap task responds by marking the swapped out process as NO_MAP, setting the residence time to 0, and marking the process as BLOCKED if it is blocked. Then the swap task goes to state 2 and tries to swap in the most eligible process (most likely, the process that caused the forced swap out).

SWAP_OUT_FAILED: This message is received when a swap out, requested by the swap task (from state 3), has failed. It is sent by MM to indicate the failure which is assumed to be caused by lack of space on the swap device. The swap task responds by marking the process as not SWAPPED and then waiting for the next CORE_IS_FREE message (state 4).

**3.4.2.1.3 Swap Out Algorithm** Processes for swap out are chosen by the swap task based on the following attributes:

- Size. The process to be swapped out should be equal to or larger than the amount of main memory needed for the swap in less the current amount of free main memory. This reduces the number of swap outs that would occur if size were not a factor. (Size is not considered in the standard UNIX swap implementations. Studies (see reference [6]) have shown that size considerations can reduce the number of swaps.) When MM returns a SWAP_IN_FAIL message, it also sends the size of additional main memory needed for the swap in. The size calculation by MM includes the total amount of existing free memory.

- Process State. The process to be swapped out must be blocked on *pause* or *wait* or be runnable. all processes currently blocked on *pause* and *wait* are marked in the MM process table. When MM returns a SWAP_IN_FAIL message, it also sends a list of all processes currently blocked on *pause* and *wait*. These processes have a higher priority for swap out than runnable processes.

- Time in Core. Thrashing is a condition in which one or more process images are continually swapped in and out. System resources are expended performing the swapping rather than performing useful work. Thrashing can occur in this design when runnable processes are swapped out because a runnable process on the swap device is immediately available for swap in. If another runnable process is chosen for swap out, then thrashing can occur. To prevent thrashing, a condition for swap out is that a process must reside in main memory for a minimal amount of time. Within that minimal amount of time, the process should get access to the CPU and progress. A potential swap out is delayed due to lack of eligible swap out candidates so a runnable process might spend a bit longer on the swap device. However, system resources are expended doing work more useful than swapping and all users experience better performance.

The following table shows the swap out priority highest to lowest:

| SIZE >= NEEDED | BLOCK TYPE | CORE RESIDENCE |
|---|---|---|
| Y | pause/wait | oldest in this category |
| Y | none | oldest above minimal |
| N | pause/wait | oldest in this category |
| N | none | oldest above minimal |

**3.4.2.1.4 Swap In Algorithm** Only runnable processes are chosen for swap in. The runnable process that has been on the swap device the longest is chosen for swap in. No minimum amount of time on the swap device is required because thrashing is minimized by the swap out algorithm.

**3.4.2.2 Clock Task** The clock task has 2 swapping related duties:

- Every second, the clock task increments the residence time of each user process. The residence time is the amount of time that the process has recently been in its current residence (main memory or on the swap device). It is set to zero whenever a process is:

  — created,

  — moved into main memory,

  — and moved onto the swap device.

  Residence time is used as a parameter in determining the eligibility of a process for swap out. Thrashing is minimized by requiring that a process remain in main memory

for a minimum amount of time before being swapped. Residence time is also an attempt, although weak, to allow equal access to the CPU for all processes. This is especially true for runnable processes that are chosen for swap out.

- The clock task checks the status of the swap task every second. If a runnable process is on the swap device and the swap task is not in a swap in state, the clock task will send a CORE_IS_FREE message to the swap task. This means that residence times have changed and that it might now be possible to find an eligible process for swap out. The swap task can then re-evaluate all processes.

**3.4.2.3 System Task** MM and FS do not have direct access to the kernel variables and kernel process table. The system task is their interface to the kernel to get data, set kernel variables, and perform miscellaneous duties. Some of the existing communications between the system task and FS and MM are used as vehicles for communicating swapping related data. The swapping related duties performed by the system task are:

- When MM performs a *fork*, it notifies the kernel to set up the kernel process table. The swapping design takes advantage of the existing *fork* communication when a *fork* swap occurs. If the *fork* message from MM indicates that a *fork* swap has occurred, the system task marks the kernel process table slot as SWAPPED, and, if the swap task is SWAP_IDLE, sends a CORE_IS_NEEDED message to the swap task.
- When MM performs an *exec*, it notifies the kernel to set up the kernel process table. If the *exec* message from MM indicates that an *exec* swap has occurred, the system task marks the kernel process table slot as SWAPPED. If the swap task is SWAP_IDLE, the

system task sends a CORE_IS_NEEDED message to the swap task to indicate that a runnable process exists on the swap device. If the *exec* message indicates that a normal *exec* has occurred and the swap task status is SWAP_IN, a CORE_IS_FREE message is sent to the swap task. This means that an *exec* has occurred that might have freed some main memory.

- When a process exits, MM notifies the kernel via the system task to clean up the process's process table slot and free it for reuse. If the swap task status is SWAP_IN, a CORE_IS_FREE message is sent to the swap task to indicate that main memory has been freed.

### 3.4.3 Memory Manager - Coordination

**3.4.3.1 Fork** The *fork* system call creates a new process by copying the current process's text, data, stack, and process table slot. The parent and child receive a different return value to enable them to identify themselves. When main memory is allocated for the child process, the allocation routine determines whether enough memory is available. If enough contiguous free memory is available, then the *fork* can continue. If free memory is available, but is not contiguous, the allocation routine compacts main memory (i.e., relocates the process images in main memory to create a large contiguous free memory space in high memory). Compaction was implemented to serve two purposes:

- reduce the number of swap outs
- simplify the swap in/out algorithms.

In either case, the memory required for the child process is allocated. However, if the required main memory is not available, the *fork* swap out is begun. It calls the *swapout* procedure (see "Swap Out Function") to copy the parent's process image to the swap device. It marks the MM process table as FKSWAPPED (*fork* swapped), notifies the kernel that a *fork* swap has occurred (see "System Task"), and notifies the kernel of the child's memory map for swap in.

**3.4.3.2 Exec** The MM *exec* system call overlays the existing program with a new program. (The program changes while the process remains.) The new program inherits the environment of the calling program. When main memory is allocated for the new program, the allocation routine determines whether enough memory is available. The memory space of the current process is included in the free memory evaluation. If enough memory is available, then the *exec* can continue. If enough memory is available, but is not contiguous, then the memory of the existing process is freed, main memory is compacted, and the memory required for the new program is allocated.

If the required main memory is not available, the *exec* swap out is begun. The new program text and data are read from the executable file on disk and written to the swap device. The stack is adjusted and also written to the swap file. The existing memory is freed, the MM process slot is marked as SWAPPED, and the kernel is notified that an *exec* swap has occurred (see "System Task").

**3.4.3.3 Paused Process Handling** There are two cases to consider:

- when any process issues the *pause* system call
- and when a *pause* swapped process becomes unpaused.

When a process issues the *pause* system call, it is blocked until the system call completes and it expects no I/O. This makes it an excellent candidate to swap out when space is needed in main memory. If, at the same time, a runnable process is on the swap device waiting to be swapped in, a CORE_IS_FREE message is sent to the swap task. This means that a process is available for swap out.

A process leaves the *pause* state upon reception of a signal. If the process is swapped out, and there are no other runnable processes on the swap device, a CORE_IS_NEEDED message is sent to the swap task. The swap task marks the kernel process table as not BLOCKED. After the process is swapped in, the expected EINTR result value is returned to the calling process.

**3.4.3.4 Wait Process Handling** There are three cases to consider:

- when a process issues the *wait* system call
- when a waiting, swapped process is unblocked by a signal
- when a waiting, swapped process receives a death of child signal.

The first two cases are the same as those described for *pause* process handling.

The death of child requires a little more handling. Death of child requires that the waiting process receive the exit status of the dead child process. This is accommodated for swapped processes by postponing the process slot cleanup of the dead process, marking it as HANGING (i.e., a zombie), freeing the dead child's memory, and saving the dead child's process slot number in the parent's process table slot. When the process is swapped in, the dead child's exit status is returned to the parent and the process slot cleanup is performed.

**3.4.3.5 Signal Handling** If a signal is sent to a swapped out process it is saved until the process has been swapped in. If the process is BLOCKED, it is marked not BLOCKED in the kernel process table and a CORE_IS_NEEDED message is sent to the swap task which will eventually start the swap in for this process. All signals received are stored in the kernel process table. When the process is swapped in and before it is marked ready to run, all pending signals are applied. Applicable signal actions are:

- terminate the process - the user process is terminated,
- ignore the signal - the signal is ignored,
- and perform a function - the user process stack is adjusted to set up a call to the signal handling function. When the process runs the signal handling function will be the first thing performed.

**3.4.3.6 Freed Memory Handling** Main core is freed by the *exit* system call and sometimes by the *exec* system call (i.e., execing a program with a smaller process memory image). The system task sys_xit and sys_exec functions determine if a runnable process is

on the swap device when either of the above occurs. If so, then they send a CORE_IS_FREE message to the swap task.

**3.4.3.7 Swap Out Function** Preparation for physical swapping of a process image to the swap device is performed by MM. This occurs two ways, *fork* swap and forced swap out. (*exec* swaps are a little different because the process image is in the executable file, not in main memory.) The swap task sends a SWAP_OUT_REQUEST message to the do_swap_out() function. The process is marked as SWAPPED. The location and size of the process segments are determined. A file name for the process is generated. The file is created and the process segments are written to the swap device. If the request is from the swap task, main memory is freed and a SWAP_OUT_COMPLETE message is sent to the swap task.

**3.4.3.8 Swap In Function** Preparation for physical swapping of a process image from the swap device is performed by MM. The swap task sends a SWAP_IN_REQUEST message to the do_swap_in() function which determines if there is enough main memory for the incoming process. If main memory is not available, then the function makes a list of all processes that are blocked on the *pause* and *wait* system calls. It sends this list along with the size of main memory needed and the SWAP_IN_FAILED message to the swap task. This aids the swap task in choosing a process for swap out.

If there is enough free main memory, then the swap in proceeds. If enough free memory exists but is not contiguous, then the allocation routine compacts memory, as described for

*fork* and *exec*, and the swap in proceeds. The memory location and size of each process segment are determined. The file name for the process is re-generated. The file is opened and the process segments are read from the swap device and copied to main memory. The process is marked as not SWAPPED and the swap file is removed. If the process was swapped out as a result of a *fork*, then a wakeup message is sent to the process just as in a normal *fork*. All signals that were received by the process while swapped out are applied. If the process was *pause* or *wait* and was interrupted by a signal, then the EINTR result is sent to the process. If the process was *wait* and was awakened by the death of a child, then the child's exit status is sent to the process and the dead process's process table slot is cleaned up. The SWAP_IN_COMPLETE message is then sent to the swap task.

### 3.4.4 File System - Swap I/O

**3.4.5 Swap Device** The swap device is the area on secondary memory that is used to store process images that have been swapped out of main memory. Swap device input/output operations should be efficient to minimize system performance degradation. Typically, space on a swap device is allocated in multiblock segments so that the swapped process images are written and read contiguously. Contiguous, multiblock reads and writes are quicker than normal file system block-by-block operations. When a process image is swapped into main memory, its storage space on the swap device is freed. Space management routines should be used to manage fragmentation that occurs on swap devices. Because the allocation scheme for the swap device differs from the normal file allocation scheme, data structures that catalog free space differ too.

The above requirements should ensure good swap device input/output performance but they also exact a price: additional code and complexity to perform and maintain that function. On a system, such as MINIX, where the main memory for user processes is limited, additional code in the operating system will further limit that memory.

This design uses an approach which sacrifices efficiency for simplicity. The existing file system is used to manage the swapping input/output operations. The swap device, therefore, consists of a normal file system directory that is protected (with normal file system protection mechanisms) from all users except root. The file system handles all swapping input/output as it handles all other file system input/output. No special contiguous, multiblock input/output operations or swap device space management procedures are required. Normal *creat, open, read, write, seek, close,* and *unlink* system calls are used. The advantages are: simple implementation, simple administration, and no large operations overhead. Disadvantages are: slow swapping input/output operations and conflicts with the file system. (When the file system is swapping, it is not free to do useful user program input/output.)

The file system manages the swap device as it does other regular files. The only change to the file system allows MM to go to the swap directory directly. The inode of the swap directory is determined and saved by the first access to the swap directory. Subsequent accesses use the saved information.

# CHAPTER 4

## 4. IMPLEMENTATION

### 4.1 Introduction

This description of the MINIX swapping implementation shows how the design elements presented in Chapter 3 are implemented. It assumes a knowledge of the MINIX version 1.3 operating system and the design described in Chapter 3. All swapping code is written in the C programming language just as most of MINIX. The code itself is commented, so the descriptions here will be brief. The description is organized into 3 parts: kernel, MM, and FS. The code is listed in the appendix[3]. New programs and functions are listed in their entirety. All C functions that have been altered are listed in their entirety. Only changes to header (.h) files are listed.

### 4.2 Kernel Code

#### 4.2.1 clock.c

**4.2.1.1 do_clocktick** Lines 64 through 69 update the kernel process table residence times for each active process by a simple increment. Residence times are reset to 0 when a process is created, moved to the swap device, and swapped into main memory. The

---

3. The original portions of the code, copyrighted by A. S. Tanenbaum, are reprinted with his permission with the restriction that reproduction be limited.

residence time variable is an unsigned integer with a maximum value of 65535. The value will turnover to 0 after about 18 hours; this is not considered to be a problem.

When there is a runnable process on the swap device, lines 70 through 75 send a CORE_IS_FREE message to the swap task as a notification that process residence times have changed. The swap task might now be able to choose a process for swap out.

### 4.2.2 proc.c

**4.2.2.1 mini_rec** Lines 53 and 54 were changed to check for a pending message from the swap task to MM. See discussion about non-blocking message transfer in "Kernel Code, swapper.c". If MM is waiting for a message, and a message from the swap task is ready, then the inform() function is called to deliver the message.

### 4.2.3 swapper.c

**4.2.3.1 swap_task** Upon entry, the init_swap() function is called to set the initial state for the swap task. Then, as with other tasks, a large case statement is entered to handle all messages that can be received.

CORE_IS_FREE (line 88) corresponds to the detailed design description. The try_to_swin() function checks for runnable processes on the swap device, picks the most eligible, and sends a swap in request to MM.

CORE_IS_NEEDED (line 94) first determines whether the message indicates that a swapped process has become unblocked. If so, the kernel process table slot is marked as

not BLOCKED. If swap_stat is SWAP_IDLE, then a swap in is attempted.

SWAP_IN_COMPL (line 105) indicates that the requested swap in has successfully completed. The process table is updated and the process is made ready to run. The swap device is checked for more runnable processes.

SWAP_IN_FAILED (line 118) indicates that the requested swap in has failed. MM has sent the size of core needed and a list of all processes blocked on the pause() and wait() system calls. MM is blocked awaiting a reply to this message. the pik_outsw() function is called to choose a process for swap out and a message is sent to MM to request the swap out. If no eligible process is chosen for swap out, the swap task responds to MM with NO_SWAP_OUT.

SWAP_OUT_COMPL (line 139) indicates that the requested swap out has completed. The NO_MAP flag is set and the BLOCKED flag is maintained by the kernel to indicate whether a swapped process is blocked or runnable (not blocked).

SWAP_OUT_FAILED (line 152) indicates that the requested swap out has failed. No action is performed until another message is received.

**4.2.3.2 try_to_swin** This function checks for runnable processes on the swap device (process table flags of SWAPPED and BLOCKED), chooses the most eligible, and requests MM to perform the swap in.

**4.2.3.3 pik_insw** This function returns the process slot number of the oldest, runnable process on the swap device.

**4.2.3.4 pik_outsw** This function returns the process slot number of the most eligible process to swap out. It does this by assigning a value to each process based on a number of process attributes. An initial priority value of 0 is assigned to each process. If the process is at least as large as the size needed, then PICKSIZE (10000) is added to the priority value. If it is blocked on the pause() or wait() system call and has been resident for the minimal interval CRESMIN (15 seconds), then PICKPWBLK (100) is assigned. If it is not blocked at all and has been resident for CRESMIN, then PICKNBLK (1) is added. If it is ineligible for swap out, PICKINELL (-20000) is added. The process with the highest priority value above 0 is returned.

**4.2.3.5 set_swapproc** A task is a high priority process and MM is a medium priority process. A process should send a message directly to a lower priority process only when it is known that the lower priority process is waiting for a message. It is not always known what MM is doing, so a non-blocking message transfer has been devised.

To send a non-blocking message to MM, such as for a SWAP_IN_REQUEST, a special kernel global structure was created called swap_proc. It contains a status which can be either MPENDING (a message is waiting to be sent to MM) or NOMES, and a message pointer. This function builds the message, sets the status to MPENDING, and sets the message pointer. The inform() function is called. If MM is waiting, the message will be

delivered immediately. If not, then it is delayed, but the swap task is not blocked. Whenever MM requests a RECEIVE for ANY, the mini_rec() function (in proc.c) checks for a non-blocking message from the kernel and the message is eventually delivered.

**4.2.4 system.c**

**4.2.4.1 do_fork** Line 39 sets residence times for all forked processes to 0, whether swapped or not.

Lines 42 through 50 are performed when a fork swap has occurred. The process table is marked and if the swap task is idle, a CORE_IS_NEEDED message is sent. The PROC1 message variable is set to 0 to indicate that the process is not blocked.

**4.2.4.2 do_newmap** When an exec swap occurs, the main memory space of the execing process is used as a data buffer. Data is copied from the executable file to this main memory area and then written to the swap file. (See MM exec.c for more details.) To set up this memory area as a buffer, exec changes the memory map of the execing process by calling the sys_newmap() function. Lines 99 through 105 check for this situation and set the process table flag to NOMAP to indicate that.

**4.2.4.3 do_exec** If an exec swap has occurred, the exswap flag is set. Lines 133 through 140 set the process table to SWAPPED and send a CORE_IS_NEEDED message to the swap task if it is SWAP_IDLE.

If an exec has occurred without a swap out and the swap task status is SWAP_IN, lines 144

through 146 send a CORE_IS_FREE message to the swap task.

**4.2.4.4 do_xit** When a process dies, main memory is freed. If the swap task status is SWAP_IN, then lines 212 through 215 send a CORE_IS FREE message to the swap task.

**4.2.4.5 do_lock** This is a new function that allows MM to turn on/off hardware interrupts. It is used when MM is performing memory compaction.

**4.2.4.6 inform** If a non-blocking message from the swap task to MM is queued, then lines 254 through 259 will send the message immediately (see set_swapproc).

### 4.3 MM Code

#### 4.3.1 alloc.c

**4.3.1.1 alloc_mem** This function has been changed, such that, if a request for contiguous memory cannot be satisfied, but enough fragmented free memory exists, then the entire memory area is compacted to satisfy the request.

**4.3.1.2 tot_hole** This is a new function that calculates the sum of all free main memory. It replaces the max_hole() function which returned the largest hole of free memory. Because compaction is used, the total free memory size is more useful than just the largest hole size.

**4.3.1.3 compact** This is a new function that performs the memory compaction. It begins by calling the sys_lock() function to turn off all system hardware interrupts so that no I/O transfers are done while a process image is being moved to another memory location. A loop is entered where the first free hole is found. The size of the process just above the hole is calculated and then the process image is copied into the hole. The process' original memory is freed and core is allocated for the process' new location. The process table memory map is adjusted. The loop repeats until there is only 1 hole remaining. The hardware interrupts are then enabled.

**4.3.2 exec.c**

**4.3.2.1 do_exec** This function has been changed to add the exec swap out feature. The call to the new_mem() function on line 63 returns EXSWAPD if main memory is not available for the execed program. New_mem() also saves a copy of the existing memory map for this process.

If a swap out is necessary, lines 93 through 185 are performed. First the swap file is created by changing to the swap directory, checking access permissions, and creating the swap file. Any failures up to this point result in an ERROR return value which indicates that the program could not be execed. The user process can handle that error any way that it chooses. Failures beyond this point result in system errors because the execing process is destroyed.

To put the execed program onto the swap device, the entire execing process memory area

(up to 2047 blocks) is converted to a temporary data buffer. The sys_newmap() function is called to notify the kernel of the change. The third parameter, TRUE, indicates that this will be a data buffer and that the system task should mark the kernel process table with the NO_MAP flag to prevent this process from being chosen to run.

The executable program is then copied to the swap device within the loop beginning at line 136. The text segment is copied into the data buffer of the execing process via the load_seg() function and then written to the swap file via the special version of the write() system call in which FS copies data directly from the user process memory area to the disk file without going through MM. After the segment has been completely copied, the file size is increased so that it is equal to the length of the segment as indicated in the MM process table memory map. This makes swap in much easier. The loop is repeated for the data segment.

The stack is created from the original stack and written to the swap file in the same manner as above.

The memory area is freed and the MM process table is marked as SWAPPED. The call to the sys_exec() function tells the kernel (swap task) about the exec and possible swap out (see Kernel Code - system.c - do_exec).

The fork swap out and forced swap out both call the swapout() function in mswap.c. They copy process images from main memory to a swap file. The exec swap does not call swapout() because it must copy the program from the executable file, transform the data

into a process image, and then write it to the swap file. In addition, the amount of memory used by the execing process is used as a buffer for the file transfer and transformation and might not be large enough for the entire new program. So it must be done piecemeal. Rather than add this complexity to the swapout() function, it was included here.

**4.3.2.2 new_mem** This function has been changed, lines 268 through 281, to check for total free memory rather than just largest contiguous free memory hole. It also includes the size of the execing process image in the free memory calculation. If memory is not available, new_mem() generates the memory map for the new program in a temporary variable and returns EXSWAPD to indicate that an exec swap out should be done.

The size of the execing process image is also calculated. If a swap out is necessary, the size will be used to determine how large the data buffer can be.

**4.3.2.3 load_seg** The new input parameter "usr" (line 340) allows the calling function to specify the user process whose memory area is being loaded. The previous version defaulted to the execing program. The new parameter is useful for the swap_in() function in mswap.c.

**4.3.3 forkexit.c**

**4.3.3.1 do_fork** This function has been changed to add the fork swap out feature and to use the memory compaction feature to reduce the number of fork swap outs. The check for free memory, lines 38 through 45, looks at total free memory rather than the largest

contiguous free memory area. If memory is not available, then a fork swap out must be done. The memory map is adjusted, if necessary. The user program controls the program stack pointer and can increase the stack size beyond the stack region. MINIX is not aware of it until the program requests more data space. When that occurs, MINIX checks the stack pointer and adjusts the stack region size in the memory map to correspond to the new stack pointer. The adjust() function verifies and adjusts, if necessary, that the stack pointer is within the stack region as defined in the memory map. The memory map is used as a measure of the process image size for swap out and swap in.

Lines 77 through 81 call the swapout() function to copy the parent's image to the swap file.

Lines 86 and 87 set the new MM process table variable to 0.

Line 109 notifies the kernel (via the system task) as to the status of the fork. If a fork swap has occurred, the system task is not notified of the childs new map nor is the reply returned to the child (because it is swapped out). It will be returned after the child is swapped in.

**4.3.3.2 mm_exit** This function has been changed, lines 164 through 173, to check for waiting parents that are swapped out. If so, then the cleanup of the process table slot and return of the exit status to the parent is delayed until the parent is swapped in. The memory occupied by the exiting process is freed in either case.

**4.3.3.3 do_wait** This function has been changed, lines 225 through 234, to check for processes on the swap device. If so, then a CORE_IS_FREE message is sent to the swap

task to indicate that the process that has just called wait() is eligible for swap out.

### 4.3.4 main.c

**4.3.4.1 main** In the call to the service functions on line 27, a dummy parameter (0) was added. This allows the service functions, specifically do_swout(), to be called with parameters by other functions (see "MM Code, do_swin").

**4.3.4.2 reply** Line 52 was added to exclude the swap task from the validation check because MM does send replies directly to the swap task and the swap task is not in the MM process table.

### 4.3.5 mswap.c This new file contains procedures for swap out and swap in.

**4.3.5.1 do_swout** This function is called from main() when the SWAP_OUT_REQUEST is received from the swap task. It is also called from do_swin() after a swap in fails and the swap task suggests a process for swap out. It starts by getting the number of and pointer to the slot of the process to be swapped out. The swapout() function is called to perform the swap out. If the swap out succeeds, the process slot is marked as SWAPPED, and if the process is blocked on the pause() or wait() system call, the PROC2 message variable is set to TRUE. The SWAP_OUT_COMPLETE message is sent to the swap task. If the swap out failed, the SWAP_OUT_FAILED message is returned.

**4.3.5.2 swapout** This function is called from the do_swout() function for a forced swap out and from the do_fork() function for a fork swap out. The big difference between the two is that the fork swap does not want to remove the parent process image from main memory, but the forced swap out does. The clear_mem variable is FALSE for the former and TRUE for the latter. The dump_core() function is called to perform the actual swapout. If it fails, an error message is returned to the caller. If it succeeds, and if it is a forced swap out, then the memory of the process is freed.

**4.3.5.3 do_swin** This function is called from main() when the swap task sends a SWAP_IN_REQUEST message. do_swin() gets the process table slot number of the process to swap in, calculates the amount of memory needed, and tries to allocate it. If memory is available, then do_swin() marks the process as not SWAPPED and calls the swap_in() function to perform the swap in. After swap in completion, many items are checked and some cleanup chores are performed.

If the process was fork swapped, then a reply of 0 is sent to indicate that it is a child of the fork.

If any signals were sent to the process while it was swapped out, then they must be applied to the process. (The actual signal handling will be done when the process is put into the RUNNING state.) During swap out, all received signals are stored in the mp_ssw_map process table variable. For each signal that was received, the sig_proc() function is called to process it.

If a process was blocked on the wait() system call and one of its child processes died, then the zombie child can now be laid to rest, its process table slot freed, and its exit status returned to its parent. All done by the cleanup() function. The slot number of the HANGING child is saved in the mp_deadchild process table variable.

If a process was blocked by a pause() or wait() system call and a signal(s) was sent to the process to wake it up, then the EINTR (i.e., system call interrupted) error value must be returned to the process.

The SWAP_IN_COMPLETE message is returned to the swap task.

If memory is not available for the swap in, then this function gathers knowledge known only by MM and sends it to the swap task. With this knowledge, the swap task can make a more informed choice of which process to select for swap out. It gathers the slot numbers of all processes in core that are blocked on the pause() and wait() system calls and stuffs them bitwise into a 32-bits variable (LONG1). It also calculates the amount of main memory needed for the swap in (PROC1).

It is necessary to freeze the user processes at this point so that they cannot progress and change their pause/wait status. So the SWAP_IN FAIL message is sent to the swap task using the send_rec() function. MM blocks on RECEIVE from the swap task. If the swap task cannot choose a process for swap out, then no action is taken. If the swap task chooses a process for swap out, the do_swout() function is called to perform the task.

**4.3.5.4 swap_in** This function is called by the do_swin() function. It generates the swap file name from the process slot number and opens the file. The memory map is changed to reflect the new base address in main memory. The gap location is initialized to zeros for security. Each segment, text, data, and stack is copied from the swap file via the loadseg() function. At completion, the swap file is removed.

### 4.3.6 signal.c

**4.3.6.1 check_sig** Lines 56 through 61 were added to handle swapped out processes to which a signal was sent. The sig_proc() function is delayed until the process is swapped in. The fact that a signal was sent is recorded bitwise in the process table variable mp_ssw_map. Note that the unpause() function is called just after this.

**4.3.6.2 unpause** Lines 117 through 121 check for swapped processes that are also paused. If so, the process table is marked as WASPWS (was pause/wait and a signal was received).

The process is now runnable, so a CORE_IS_NEEDED message is sent to the swap task with the process table slot number of the awakened process. The swap task will mark the process table as not BLOCKED and attempt to swap in the process. Lines 129 through 133 accomplish the same thing for waiting processes.

**4.3.6.3 do_pause** Lines 85 through 92 send a CORE_IS_FREE message to the swap task to indicate that a process has blocked itself and is available for swap out if necessary.

**4.3.6.4 dump_core** This function, as its name implies, was used only for writing the process image to a file on disk. It has been altered to serve another purpose: swap a process image out to the swap device. It just so happens that these two functions are similar enough to handle them in a general manner. Changes were made throughout the function so it will be described in its entirety.

The input parameter "type" is 0 for a dump core request, 1 for a swap out, and 2 for a fork swap out. The function changes to the working directory of the user process for dump core or the swap device for swap outs. The file is checked for write access permission. For swap out, the swap file name is generated from the process table slot number. The file is created; if it fails, an error is returned.

The memory map is checked for accuracy (i.e., is the stack pointer within the stack region?) and adjusted, if necessary.

For core dumps, the memory map is written to the core file. Each process segment, text, data, and stack is written to the core file or swap file (as appropriate). The write() system call to FS is special in that the data is copied directly from the user memory area and written to disk without going through the MM.

### 4.4 FS Code

#### 4.4.1 main.c

**4.4.1.1 fsinit** A new variable, swap_node is declared and initialized to NIL on lines 10 and 29. It is used to store the inode of the swap directory.

### 4.4.2 stadir.c

**4.4.2.1 do_chdir** Lines 38 through 49 were added to allow MM to quickly change to the swap directory when the cd_flag is 2. If the directory has not been opened yet, then the change() function is called to get the inode and store it in the swap_node variable.

**4.4.2.2 change** Lines 80 through 83 explicitly set the user_path to the name_ptr if the cd_flag is 2.

# CHAPTER 5

## 5. CONCLUSIONS

### 5.1 Results

The implementation of swapping satisfies the requirements of Chapter 2:

- It successfully extends the limited memory of the system. The CPU is more fully utilized and user frustration is reduced.
- All existing functions are maintained; no new functionality was added.
- The MINIX structure is maintained; related extensions are suggested below.
- Installation is simple. It requires installing a protected directory "/usr/swap". Maintenance requires ensuring that enough secondary memory is available for the swap device.
- Performance is acceptable. It is not degraded when swapping is not in use and shows acceptable degradation when in light use. Table 5-1 shows benchmarks of system performance under various conditions. The benchmarks are based on compiles of the MINIX kernel, MM, and FS using the MINIX supplied compiler and makefiles. Note that there are no perceptible compile time differences between MINIX version 1.3 without swapping and version 1.3 with swapping when no swapping occurs. When swapping occurs, the performance starts to suffer. The AT class machine takes about 23% more time to concurrently compile the kernel and MM while the XT class only

requires about 11% more time. This is considered light to medium swapping and because the compiles are CPU intensive, it is acceptable. For heavier swapping, the AT class machine takes about 52% more time to concurrently compile the kernel, MM, and FS. This is probably exceeding the capacity of the CPU and is a hardware problem.

The size of the MINIX process memory image increased by 10% (about 12K bytes). (Part of the increase, about 13%, is a result of increasing the number of process table slots from 16 to 24 to allow more processes to be active.)

Response time for interactive processes suffers when CPU bound processes are running. It becomes more prevalent with swapping because more processes can be run concurrently. The response degradation occurs because MINIX assigns the same priority to all user processes. A new process scheduler algorithm is required to remedy this.

### 5.2 Improvements and Further Development

The implementation provides an opportunity to devise, experiment, and test various scheduling and swapping algorithms. In addition, the implementation can be extended in other ways:

- User process priorities can be established to assign interactive processes a higher user priority to provide immediate response.
- Better swap out/in algorithms can be implemented, possibly integrating process scheduling with process swapping.

- Shared text can be implemented to make swapping more efficient.
- A real *brk()* system call can now be implemented in MINIX.
- The exec swap out can be optimized to eliminate a read and write of the incoming program.
- A more efficient swap device can be implemented.

| OS | COMPILE TASK | HARDWARE | |
|---|---|---|---|
| | | AT | XT |
| MINIX 1.3 v | kernel | 7:43 | |
| | FS | 5:43 | |
| | MM | 3:22 | |
| | TOTAL | 16:48 | |
| MINIX 1.3v w/swapping not used | kernel | 7:45 | 49:09 |
| | FS | 5:53 | |
| | MM | 3:24 | 20:02 |
| | TOTAL | 17:02 | 69:11 |
| MINIX 1.3 v w/swapping in use | kernel & MM | 13:44 | 75:42 |
| | kernel, MM, & FS | 25:38 | |

**TABLE 5-1.** MINIX NON-SWAPPING vs SWAPPING BENCHMARKS

**BIBLIOGRAPHY**

[1] Tanenbaum, A. S., "Operating Systems, Design and Implementation", Prentice-Hall, Inc., 1987

[2] Bach, M. J., "The Design of the UNIX Operating System", Prentice-Hall, Inc., 1986

[3] Lauer, H.C., Needham, R.M., "On the Duality of Operating System Structures" in Proc. Second International Symposium on Operating Systems, IRIA, Oct 1978, reprinted in Operating Systems Review, 13,2, April 1979, pp 3-19

[4] Peachey, D. R., Bunt, R. B., Williamson, L.L., Brecht, T. B., "An Experimental Investigation on Scheduling Strategies for UNIX", Performance Evaluation Review, V12, 3 (August 1984), pp 158-166

[5] Abdallah, M. S., "An Investigation of the Swapping Process in the UNIX Operating System", MSc. Thesis, Department of Computational Science, University of Saskatchewan, July 1982.

[6] Peachey, R. B., Williamson, L.L., Bunt, R. B., "Taming the UNIX Scheduler", Proceedings CMG XV, San Francisco, Dec 1984

# APPENDIX A - MODIFICATIONS TO KERNEL CODE

# Appendix A-2 - MODIFICATIONS TO KERNEL CODE

Jul 6 14:30 1989 KERNEL.H Page 1

```
                const.h

>   /* swap_task defines */
>   #define SWAP_IDLE 1         /* nothing to swap in or out */
>   #define SWAP_IN             2       /* runnable process on swap device */
>   #define SWAP_OUT 3          /* swap_task has ordered a swap out */
>   #define NOMES               4       /* No message is waiting for MM */
>   #define MPENDING 5          /* Message is waiting for MM */


                glo.h

>   /* trace display */
>   EXTERN int Dflag;           /* if == 0, no display, else auto display */
>   EXTERN int swap_stat;
>   EXTERN struct sw_mm_mes swap_proc;


                proc.h

>   unsigned res_time;          /* residence time in seconds (core, swap) */
>   #define SWAPPED             040     /* process is on swap device */
>   #define BLOCKED             0100    /* swapped proc is blocked */
>   #define STICKY              0200    /* proc has sticky bit set */


                type.h

>   PUBLIC struct sw_mm_mes {
>               int status;
>               message *ms;
>   };
```

# Appendix A-3 - MODIFICATIONS TO KERNEL CODE



```
/*===========================================================================*
 *                              do_clocktick                                 *
 *===========================================================================*/
PRIVATE do_clocktick()
{
/* This routine is called on every clock tick. */

  register struct proc *rp;
  register int t, proc_nr;
  extern int pr_busy, pcount, cum_count, prev_ct;

  /* To guard against race conditions, first copy 'lost_ticks' to a local
   * variable, add this to 'realtime', and then subtract it from 'lost_ticks'.
   */
  t = lost_ticks;                 /* 'lost_ticks' counts missed interrupts */
  realtime += t + 1;              /* update the time of day */
  lost_ticks -= t;                /* these interrupts are no longer missed */

  if (next_alarm <= realtime) {
        /* An alarm may have gone off, but proc may have exited, so check. */
        next_alarm = MAX_P_LONG;  /* start computing next alarm */
        for (rp = &proc[0]; rp < &proc[NR_TASKS+NR_PROCS]; rp++) {
                if (rp->p_alarm != (real_time) 0) {
                        /* See if this alarm time has been reached. */
                        if (rp->p_alarm <= realtime) {
                                /* A timer has gone off. If it is a user proc,
                                 * send it a signal. If it is a task, call the
                                 * function previously specified by the task.
                                 */
                                proc_nr = rp - proc - NR_TASKS;
                                if (proc_nr >= 0)
                                        cause_sig(proc_nr, SIGALRM);
                                else
                                        (*watch_dog[-proc_nr])();
                                rp->p_alarm = 0;
                        }

                        /* Work on determining which alarm is next. */
                        if (rp->p_alarm != 0 && rp->p_alarm < next_alarm)
                                next_alarm = rp->p_alarm;
                }
        }
  }

  accounting();                   /* keep track of who is using the cpu */

  /* If input characters are accumulating on an RS232 line, process them. */
  if (flush_flag) {
        t = (int) realtime;       /* only low-order bits matter */
        if ( (t & FLUSH_MASK) == 0) rs_flush(); /* flush tty input */
  }

  /* If a user process has been running too long, pick another one. */
```

# Appendix A-4 - MODIFICATIONS TO KERNEL CODE

Jul 6 15:13 1989 CLOCK.C Page 2

```
        if (--sched_ticks == 0) {
                if (bill_ptr == prev_ptr) sched(); /* process has run too long */
                sched_ticks = SCHED_RATE;               /* reset quantum */
                prev_ptr = bill_ptr;                    /* new previous process */

                /* Check if printer is hung up, and if so, restart it. */
                if (pr_busy && pcount > 0 && cum_count == prev_ct) pr_char();
                prev_ct = cum_count;/* record # characters printed so far */
        }

>       /* if residence update time, then update all residence times */
>       if (resd_ticks <= realtime) {
>               resd_ticks = realtime + RES_RATE;
>               for (rp = proc_addr(LOW_USER); rp < &proc[NR_TASKS+NR_PROCS]; rp++)
>                       if (rp->p_flags != P_SLOT_FREE)
>                               rp->res_time++;
>               if(swap_stat == SWAP_IN) {
>                   /* notify swap task that residence times have changed */
>                   mess.m_source = CLOCK;
>                   mess.m_type = CORE_IS_FREE;
>                   send(SWAP_TASK, &mess);
>               }
                if(seconds++ >= 5) {
                        seconds = 0;
                        if(Dflag)
                                a_dmp();
                }
        }
    }
```

# Appendix A-5 - MODIFICATIONS TO KERNEL CODE



```
/*===========================================================================*
 *                              mini_rec                                     *
 *===========================================================================*/
PRIVATE int mini_rec(caller, src, m_ptr)
int caller;                     /* process trying to get message */
int src;                        /* which message source is wanted (or ANY) */
message *m_ptr;                         /* pointer to message buffer */
{
/* A process or task wants to get a message.  If one is already queued,
 * acquire it and deblock the sender.  If no message from the desired source
 * is available, block the caller.  No need to check parameters for validity.
 * Users calls are always sendrec(), and mini_send() has checked already.
 * Calls from the tasks, MM, and FS are trusted.
 */

 extern struct sw_mm_mes swap_proc;
 register struct proc *caller_ptr, *sender_ptr, *previous_ptr;
 int sender;

 caller_ptr = proc_addr(caller);  /* pointer to caller's proc struct */

 /* Check to see if a message from desired source is already available. */
 sender_ptr = caller_ptr->p_callerq;
 if ((caller_ptr->p_flags & SENDING) == 0) {
    while (sender_ptr != NIL_PROC) {
          sender = sender_ptr - proc - NR_TASKS;
          if (src == ANY || src == sender) {
                  /* An acceptable message has been found. */
                  cp_mess(sender, sender_ptr->p_map[D].mem_phys,
                    sender_ptr->p_messbuf, caller_ptr->p_map[D].mem_phys, m_ptr);
                  sender_ptr->p_flags &= SENDING;          /* deblock sender */
                  if (sender_ptr->p_flags == 0) ready(sender_ptr);
                  if (sender_ptr == caller_ptr->p_callerq)
                          caller_ptr->p_callerq = sender_ptr->p_sendlink;
                  else
                          previous_ptr->p_sendlink = sender_ptr->p_sendlink;
                  return(OK);
          }
          previous_ptr = sender_ptr;
          sender_ptr = sender_ptr->p_sendlink;
    }
 }

/* No suitable message is available.  Block the process trying to receive. */
caller_ptr->p_getfrom = src;
caller_ptr->p_messbuf = m_ptr;
if (caller_ptr->p_flags == 0) unready(caller_ptr);
caller_ptr->p_flags |= RECEIVING;

/* If MM has just blocked and there are kernel signals pending, now is the
 * time to tell MM about them, since it will be able to accept the message.
 */
if( ((sig_procs > 0) || (swap_proc.status == MPENDING)) &&
```

Jul 6 15:13 1989 PROC.C Page 2

```
>           (caller == MM_PROC_NR && src == ANY) ) {
                    inform();
                    pick_proc();
            }
            return(OK);
        }
```

# Appendix A-7 - MODIFICATIONS TO KERNEL CODE



```
/* This file contains the code and data for the swapper task. It
 * has a single entry point, swap_task(). It accepts six message
 * types:
 *
 *  CORE_IS_FREE:    memory has been released or can possibly be freed
 *  CORE_IS_NEEDED:  runnable process is on swap device
 *  SWAP_IN_COMPL:   swap_task request has succeeded
 *  SWAP_IN_FAILED:  swap_task request has failed
 *  SWAP_OUT_COMPL:  swap_task or MM request has succeeded
 *  SWAP_OUT_FAILED: swap_task request has failed
 *
 * The input message is format m1. The parameters are:
 *
 *  m_type           PROC1     PROC2     PID     MEM_PTR   UTILITY
 * ---------------------------------------------------------------
 * | CORE_IS_FREE    |         |         |        |        |        |
 * |-----------------+---------+---------+--------+--------+--------+
 * | CORE_IS_NEEDED  |unblocked|         |        |        |        |
 * |-----------------+---------+---------+--------+--------+--------+
 * | SWAP_IN_COMPL   |swappedin|         |        |        |        |
 * |-----------------+---------+---------+--------+--------+--------+
 * | SWAP_OUT_COMPL  |swapedout| blocked |        |        |        |
 * |-----------------+---------+---------+--------+--------+--------+
 * | SWAP_OUT_FAILED | proc no |         |        |        |        |
 * ---------------------------------------------------------------
 *
 * The input message is format m2. The parameters are:
 * ----------------+---------+---------+--------+--------+----------
 * | SWAP_IN_FAILED  |sizeofcor|pausewait|        |        |        |
 * ----------------+---------+---------+--------+--------+----------
 */

#include "../h/const.h"
#include "../h/type.h"
#include "../h/callnr.h"
#include "../h/com.h"
#include "../h/error.h"
#include "../h/signal.h"
#include "const.h"
#include "type.h"
#include "glo.h"
#include "proc.h"

/* constant definitions */
#define NOP          1          /* nothing in progress */
#define SW_INP       2          /* swap-in in progress */
#define SW_OUTP      3          /* swap-out in progress */

/* swap out algotithm definitions */
#define PICKSIZE     10000      /* size is >= size needed */
#define PICKSTICK    1000       /* sticky bit is not set */
#define PICKPWBLK            100     /* proc is PAUSE/WAIT */
#define PICKSRBLK    10         /* proc is blocked on both S & R */
```

```
#define PICKNBLK      1      /* proc is not blocked at all */
#define PICKINELL     -20000 /* proc is inelligible for swap out */

#define CRESMIN              15      /* min residence time in seconds */
#define LONG1                m2_l1   /* message slot to carry long */

/* swapper task variables */
PRIVATE int inprogres;
PRIVATE message mes, mms;


/*===========================================================================*
 *                              swap_task                                    *
 *===========================================================================*/
PUBLIC swap_task()
{
/* Main program of swap task. It determines which of the 6 possible
 * calls this is by looking at 'mes.m_type'.  Then it dispatches.
 */
 struct proc *rp;
 int opcode, pnum;
 long bm;
 phys_clicks sizeneed;
 phys_bytes src_phys, dst_phys;
 vir_bytes ptr;

 init_swap();                          /* initialize swap tables */

 /* Main loop of the swap task.  Get work, process it, sometimes reply. */
 while (TRUE) {
   receive(ANY, &mes);                 /* go get a message */
   opcode = mes.m_type;                /* extract the message type */

   switch (opcode) {
	case CORE_IS_FREE:
	    if(swap_stat == SWAP_IN) && (inprogres == NOP) {
			try_to_swin();
	    }
	    break;

	case CORE_IS_NEEDED:
	    if(mes.PROC1 != 0) {
		/* PROC1 on swap device just became unblocked */
		rp = proc_addr(mes.PROC1);
		rp->p_flags &=  BLOCKED;
	    }
	    if(swap_stat == SWAP_IDLE) {
		try_to_swin();
	    }
	    break;

	case SWAP_IN_COMPL:
	    if(inprogres == SW_INP) {
```

Jul 6 15:13 1989 SWAPPER.C Page 3

```
				inprogres = NOP;
				pnum = mes.PROC1;
				rp = proc_addr(pnum);
				rp->p_flags &= SWAPPED;
				rp->res_time = 0;
				if (rp->p_flags == 0) ready(rp);
				try_to_swin();
			} else
				printf("S_I_C: swapin NOT in progress0);
			break;

		case SWAP_IN_FAILED:
			if(inprogres == SW_INP) {
				inprogres = NOP;
				sizeneed = (phys_clicks)mes.PROC1;
				bm = mes.LONG1;
				mms.m_source = SWAP_TASK;
				if( (pnum = pik_outsw(bm, sizeneed)) != 0) {
					rp = proc_addr(pnum);
					rp->p_flags |= SWAPPED;
					unready(rp);
					inprogres = SW_OUTP;
					mms.m_type = SWAP_OUT_REQ;
					mms.PROC1 = pnum;
			} else {
					mms.m_type = NO_SWAP_OUT;
				}
				send(MM_PROC_NR, &mms);
			} else
				printf("S_I_F: swap in NOT in progress0);
			break;

		case SWAP_OUT_COMPL:
			if(inprogres == SW_OUTP) {
				inprogres = NOP;
				rp = proc_addr(mes.PROC1);
				rp->p_flags |= NO_MAP;
				if(mes.PROC2)
					rp->p_flags |= BLOCKED;
				rp->res_time = 0;
				try_to_swin();
			} else
				printf("S_O_C: swap out NOT in progress0);
			break;

		case SWAP_OUT_FAILED:
			if(inprogres == SW_OUTP) {
				rp = proc_addr(mes.PROC1);
				rp->p_flags &= SWAPPED;
				inprogres = NOP;
				if (rp->p_flags == 0) ready(rp);
			} else
				printf("S_O_F: swap out NOT in progress0);
```

```
                break;

            default: panic("swap task got bad message", mes.m_type);
                break;
        }
    }
}

init_swap()
{
            inprogres = NOP;
            swap_proc.status = NOMES;
            swap_stat = SWAP_IDLE;
}

/* set up non-blocking message transfer to MM */
set_swapproc(type, num)
int type, num;
{
            mms.m_source = SWAP_TASK;
            mms.m_type = type;
            mms.PROC1 = num;
            swap_proc.ms = &mms;
            swap_proc.status = MPENDING;
            if( ((proc[NR_TASKS + MM_PROC_NR].p_flags & RECEIVING) == 0) ||
               (proc[NR_TASKS + MM_PROC_NR].p_getfrom != ANY) )
                    return;
            inform();
}

try_to_swin()
{
int proc;
            if( (proc = pik_insw()) == 0) {
               swap_stat = SWAP_IDLE;
            } else {
               inprogres = SW_INP;
               swap_stat = SWAP_IN;
               set_swapproc(SWAP_IN_REQ, proc);
            }
}

pik_insw()
{
/* returns the proc slot # of the most elligible proc on swap device */
/* to swap in.   If none are elligible, then 0 is returned */
struct proc *rp;
struct proc *stkrp;
struct proc *nostkrp;
unsigned stkres, nostkres;
int pick;

  stkrp = proc_addr(0);
```

```
   nostkrp = proc_addr(0);
   stkres = 0;
   nostkres = 0;

   for(rp = proc_addr(INIT_PROC_NR +1); rp < proc_addr(NR_PROCS); rp++) {
           if(rp->p_flags & P_SLOT_FREE) continue;
           if( (rp->p_flags & SWAPPED)  &&
              ((rp->p_flags & BLOCKED) == 0) ) {
                   if(rp->p_flags & STICKY) {
                      if(rp->res_time >= stkres) {
                           stkrp = rp;
                           stkres = rp->res_time;
                      }
                   } else if(rp->res_time >= nostkres) {
                           nostkrp = rp;
                           nostkres = rp->res_time;
                   }
           }
   }
   if(stkrp != proc_addr(0))
           pick = (int)(stkrp - proc_addr(0));
   else
           pick = (int)(nostkrp - proc_addr(0));
   return(pick);
}


pik_outsw(map, sizeneed)
long map;
int sizeneed;
{
/* returns the proc slot # of the most elligible proc in core */
/* to swap out.  If none are elligible, then 0 is returned */

struct proc *rp, *hrp;
int priority, hpriority;
unsigned hres;

  hpriority = 0;
  hres = 0;
  hrp = proc_addr(0);
  for(rp = proc_addr(INIT_PROC_NR +1); rp < proc_addr(NR_PROCS); rp++) {
          if(rp->p_flags & (P_SLOT_FREE | SWAPPED)) continue;
          priority = 0;
          if( (rp->p_map[S].mem_phys +
              rp->p_map[S].mem_len  -
              rp->p_map[T].mem_phys) >= sizeneed)
                  priority += PICKSIZE;
          if( (rp->p_flags & STICKY) == 0)
                  priority += PICKSTICK;
          priority += blocktype(rp, map);
          if( (priority > hpriority) ||
             (priority == hpriority && rp->res_time > hres) ) {
```

```
				hpriority = priority;
				hres = rp->res_time;
				hrp = rp;
			}
	}
	if(hpriority == 0)
		return(0);
	return( (int)(hrp - proc_addr(0)));
}


/* determine if process is blocked or pause/wait, SEND & RECEIVE, */
/* or not at all */
blocktype(rp, map)
struct proc *rp;
long map;
{

		if( (map >> (rp - proc_addr(0)) & 1) &&
		    (rp->res_time >= CRESMIN) )
			return(PICKPWBLK);
		if( (rp->p_flags & (SENDING | RECEIVING) == 0) &&
		    (rp->res_time >= CRESMIN) )
			return(PICKNBLK);
		if( (rp->p_flags & (SENDING | RECEIVING) == (SENDING | RECEIVING)) &&
		    (rp->res_time >= CRESMIN) )
			return(PICKINELL);
		return(PICKINELL);
}
```

# Appendix A-13 - MODIFICATIONS TO KERNEL CODE

Jul 6 15:13 1989 SYSTEM.C Page 1

```
/*===========================================================================*
 *                              do_fork                                      *
 *===========================================================================*/
PRIVATE int do_fork(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
/* Handle sys_fork(). 'k1' has forked. The child is 'k2'. */

  register struct proc *rpc;
  register char *sptr, *dptr;   /* pointers for copying proc struct */
  int k1;                       /* number of parent process */
  int k2;                       /* number of child process */
  int pid;                      /* process id of child */
  int bytes;                    /* counter for copying proc struct */
  int fkswap;                           /* TRUE, if fork has swapped out child */

  k1 = m_ptr->PROC1;                    /* extract parent slot number from msg */
  k2 = m_ptr->PROC2;                    /* extract child slot number */
  pid = m_ptr->PID;             /* extract child process id */
  fkswap = (int)m_ptr->UTILITY;         /* extract swap status of child */

  if (k1 < 0 || k1 >= NR_PROCS || k2 < 0 || k2 >= NR_PROCS)return(E_BAD_PROC);
  rpc = proc_addr(k2);

  /* Copy parent 'proc' struct to child. */
  sptr = (char *) proc_addr(k1); /* parent pointer */
  dptr = (char *) proc_addr(k2); /* child pointer */
  bytes = sizeof(struct proc);          /* # bytes to copy */
  while (bytes--) *dptr++ = *sptr++;    /* copy parent struct to child */

  rpc->p_flags |= NO_MAP;       /* inhibit the process from running */
  rpc->p_flags &= PENDING;      /* only one in group should have PENDING */
  rpc->p_pending = 0;
  rpc->p_pid = pid;             /* install child's pid */
  rpc->p_reg[RET_REG] = 0;      /* child sees pid = 0 to know it is child */

  rpc->user_time = 0;           /* set all the accounting times to 0 */
  rpc->sys_time = 0;
> rpc->res_time = 0;
  rpc->child_utime = 0;
  rpc->child_stime = 0;
> if(fkswap) {
>   rpc->p_flags |= SWAPPED;
>   if(swap_stat == SWAP_IDLE) {
>     mess.m_source = SYSTASK;
>     mess.m_type = CORE_IS_NEEDED;
>     mess.PROC1 = 0;
>     send(SWAP_TASK, &mess);
>   }
> }
  return(OK);
}
```

```

/*===========================================================================*
 *                              do_newmap                                    *
 *===========================================================================*/
PRIVATE int do_newmap(m_ptr)
message *m_ptr;                        /* pointer to request message */
{
/* Handle sys_newmap(). Fetch the memory map from MM. */

  register struct proc *rp, *rsrc;
  phys_bytes src_phys, dst_phys, pn;
  vir_bytes vmm, vsys, vn;
  int caller;                  /* whose space has the new map (usually MM) */
  int k;                       /* process whose map is to be loaded */
  int old_flags;               /* value of flags before modification */
  struct mem_map *map_ptr;     /* virtual address of map inside caller (MM) */
  int util;                    /* TRUE, if exec uses core as a buffer */

  /* Extract message parameters and copy new memory map from MM. */
  caller = m_ptr->m_source;
  k = m_ptr->PROC1;
  map_ptr = (struct mem_map *) m_ptr->MEM_PTR;
  util = (int)m_ptr->UTILITY;   /* extract swap status of child */
  if (k < -NR_TASKS || k >= NR_PROCS) return(E_BAD_PROC);
  rp = proc_addr(k);           /* ptr to entry of user getting new map */
  rsrc = proc_addr(caller);     /* ptr to MM's proc entry */
  vn = NR_SEGS * sizeof(struct mem_map);
  pn = vn;
  vmm = (vir_bytes) map_ptr;   /* careful about sign extension */
  vsys = (vir_bytes) rp->p_map; /* again, careful about sign extension */
  if ( (src_phys = umap(rsrc, D, vmm, vn)) == 0)
        panic("bad call to sys_newmap (src)", NO_NUM);
  if ( (dst_phys = umap(proc_addr(SYSTASK), D, vsys, vn)) == 0)
        panic("bad call to sys_newmap (dst)", NO_NUM);
  phys_copy(src_phys, dst_phys, pn);

#ifdef i8088
  /* On 8088, set segment registers. */
  rp->p_reg[CS_REG] = rp->p_map[T].mem_phys;     /* set cs */
  rp->p_reg[DS_REG] = rp->p_map[D].mem_phys;     /* set ds */
  rp->p_reg[SS_REG] = rp->p_map[D].mem_phys;     /* set ss */
  rp->p_reg[ES_REG] = rp->p_map[D].mem_phys;     /* set es */
#endif

  /* don't make process ready if core is being used as an I/O buffer */
> if(! util) {
>     old_flags = rp->p_flags;    /* save the previous value of the flags */
>     rp->p_flags &= NO_MAP;
>     if (old_flags != 0 && rp->p_flags == 0) ready(rp);
> } else {
>     rp->p_flags |= NO_MAP;
> }
  return(OK);
```

```
}


/*===========================================================================*
 *                              do_exec                                      *
 *===========================================================================*/
PRIVATE int do_exec(m_ptr)
message *m_ptr;                         /* pointer to request message */
{
/* Handle sys_exec().  A process has done a successful EXEC. Patch it up. */

  register struct proc *rp;
  int k;                        /* which process */
  int *sp;                      /* new sp */
  int exswap;                           /* TRUE, if exec has swapped out new proc */

  k = m_ptr->PROC1;                     /* 'k' tells which process did EXEC */
  sp = (int *) m_ptr->STACK_PTR;
  exswap = (int)m_ptr->UTILITY;         /* extract swap status of new proc */
  if (k < 0 || k >= NR_PROCS) return(E_BAD_PROC);
  rp = proc_addr(k);
  rp->p_sp = sp;                /* set the stack pointer */
  rp->p_pcpsw.pc = (int (*)()) 0; /* reset pc */
  rp->p_alarm = 0;              /* reset alarm timer */
  rp->p_flags &=  RECEIVING; /* MM does not reply to EXEC call */

> if(exswap) {
>         rp->p_flags |= SWAPPED;
>         if(swap_stat == SWAP_IDLE) {
>            mess.m_source = SYSTASK;
>            mess.m_type = CORE_IS_NEEDED;
>            mess.PROC1 = 0;
>            send(SWAP_TASK, &mess);
>         }
> } else if(swap_stat == SWAP_IN) {
> /* notify swapper that core has been freed */
>   mess.m_source = SYSTASK;
>   mess.m_type = CORE_IS_FREE;
>   send(SWAP_TASK, &mess);
> }

  if (rp->p_flags == 0) ready(rp);
  if(exswap)
          set_name(k, (char *)0);       /* erase command string from F1 display */
  else
          set_name(k, (char *)sp);      /* save command string for F1 display */

  return(OK);
}


/*===========================================================================*
 *                              do_xit                                       *
```

```
 *===========================================================================*/
PRIVATE int do_xit(m_ptr)
message *m_ptr;				/* pointer to request message */
{
/* Handle sys_xit(). A process has exited. */

  register struct proc *rp, *rc;
  struct proc *np, *xp;
  int parent;				/* number of exiting proc's parent */
  int proc_nr;				/* number of process doing the exit */

  parent = m_ptr->PROC1;	/* slot number of parent process */
  proc_nr = m_ptr->PROC2;	/* slot number of exiting process */
  if (parent < 0 || parent >= NR_PROCS || proc_nr < 0 || proc_nr >= NR_PROCS)
	return(E_BAD_PROC);
  rp = proc_addr(parent);
  rc = proc_addr(proc_nr);
  rp->child_utime += rc->user_time + rc->child_utime;/* accum child times */
  rp->child_stime += rc->sys_time + rc->child_stime;
  rc->p_alarm = 0;		/* turn off alarm timer */
  if (rc->p_flags == 0) unready(rc);
  set_name(proc_nr, (char *) 0); /* disable command printing for F1 */

  /* If the process being terminated happens to be queued trying to send a
   * message (i.e., the process was killed by a signal, rather than it doing an
   * EXIT), then it must be removed from the message queues.
   */
  if (rc->p_flags & SENDING) {
	/* Check all proc slots to see if the exiting process is queued. */
	for (rp = &proc[0]; rp < &proc[NR_TASKS + NR_PROCS]; rp++) {
		if (rp->p_callerq == NIL_PROC) continue;
		if (rp->p_callerq == rc) {
			/* Exiting process is on front of this queue. */
			rp->p_callerq = rc->p_sendlink;
			break;
		} else {
			/* See if exiting process is in middle of queue. */
			np = rp->p_callerq;
			while ( ( xp = np->p_sendlink) != NIL_PROC)
				if (xp == rc) {
					np->p_sendlink = xp->p_sendlink;
					break;
				} else {
					np = xp;
				}
		}
	}
  }
  if (rc->p_flags & PENDING) --sig_procs;
  rc->p_flags = P_SLOT_FREE;

> /* notify swapper that core has been freed */
> if(swap_stat == SWAP_IN) {
```

# Appendix A-17 - MODIFICATIONS TO KERNEL CODE



```
>     mess.m_type = CORE_IS_FREE;
>     send(SWAP_TASK, &mess);
>   }

  return(OK);
}


/*===========================================================================*
 *                              do_lock                                      *
 *===========================================================================*/
> PRIVATE int do_lock(m_ptr)
> message *m_ptr;                       /* pointer to request message */
> {
> /* Handle sys_lock or restore request from MM */
>
>   if(m_ptr->LOCK_RES == LOCK) {
>     m_ptr->PSW = (int) lock();
>   } else {
>     restore( (unsigned) m_ptr->PSW);
>   }
>   return(OK);
> }
>
>
/*===========================================================================*
 *                              inform                                       *
 *===========================================================================*/
PUBLIC inform()
{
/* When a signal is detected by the kernel (e.g., DEL), or generated by a task
 * (e.g. clock task for SIGALRM), cause_sig() is called to set a bit in the
 * p_pending field of the process to signal.  Then inform() is called to see
 * if MM is idle and can be told about it.  Whenever MM blocks, a check is
 * made to see if 'sig_procs' is nonzero; if so, inform() is called.
 */

  register struct proc *rp;

  /* MM is waiting for new input.  Find a process with pending signals. */
        /* does swapper want to send message to MM now? */
>       if(swap_proc.status == MPENDING) {
>               if (mini_send(SWAP_TASK, MM_PROC_NR, swap_proc.ms) != OK)
>                       panic("can't inform MM", NO_NUM);
>               swap_proc.status = NOMES;
>               return;
>       }
        for (rp = proc_addr(0); rp < proc_addr(NR_PROCS); rp++)
        if (rp->p_flags & PENDING) {
                m.m_type = KSIG;
                m.PROC1 = rp - proc - NR_TASKS;
                m.SIG_MAP = rp->p_pending;
                sig_procs--;
```

```
			if (mini_send(HARDWARE, MM_PROC_NR, &m) != OK)
				panic("can't inform MM", NO_NUM);
			rp->p_pending = 0;	/* the ball is now in MM's court */
			rp->p_flags &=  PENDING;
			if (rp->p_flags == 0) ready(rp);
			return;
		}
	}
```

# Appendix A-19 - MODIFICATIONS TO KERNEL CODE

Jul 6 15:13 1989 TABLE.C Page 1

```
/* The object file of "table.c" contains all the data. In the *.h files,
 * declared variables appear with EXTERN in front of them, as in
 *
 *    EXTERN int x;
 *
 * Normally EXTERN is defined as extern, so when they are included in another
 * file, no storage is allocated. If the EXTERN were not present, but just
 * say,
 *
 *    int x;
 *
 * then including this file in several source files would cause 'x' to be
 * declared several times. While some linkers accept this, others do not,
 * so they are declared extern when included normally. However, it must
 * be declared for real somewhere. That is done here, by redefining
 * EXTERN as the null string, so the inclusion of all the *.h files in
 * table.c actually generates storage for them. All the initialized
 * variables are also declared here, since
 *
 * extern int x = 4;
 *
 * is not allowed. If such variables are shared, they must also be declared
 * in one of the *.h files without the initialization.
 */

#include "../h/const.h"
#include "../h/type.h"
#include "../h/com.h"
#include "const.h"
#include "type.h"
#undef   EXTERN
#define  EXTERN
#include "glo.h"
#include "proc.h"
#include "tty.h"

extern int sys_task(), clock_task(), mem_task(), floppy_task(),
>          winchester_task(), tty_task(), printer_task(), swap_task();

#ifdef AM_KERNEL
extern int amoeba_task();
extern int amint_task();
#endif

/* The startup routine of each task is given below, from -NR_TASKS upwards.
 * The order of the names here MUST agree with the numerical values assigned to
 * the tasks in ../h/com.h.
 */
> #define    SMALL_STACK     512

#define     TTY_STACK       SMALL_STACK

> #define SWAP_STACK        SMALL_STACK
```

```

#defioe PRINTER_STACK     SMALL_STACK
#define    WINCH_STACK    SMALL_STACK
#define    FLOP_STACK     SMALL_STACK
#define    MEM_STACK      SMALL_STACK
#define    CLOCK_STACK    SMALL_STACK
#define    SYS_STACK      SMALL_STACK



#ifdef AM_KERNEL
#          define    AMINT_STACK            SMALL_STACK
#          define    AMOEBA_STACK           1532
#          define    AMOEBA_STACK_SPACE     (AM_NTASKS*AMOEBA_STACK + AMINT_STACK)
#else
#          define    AMOEBA_STACK_SPACE     0
#endif

#define    TOT_STACK_SPACE                 (TTY_STACK + AMOEBA_STACK_SPACE + \
                                      SWAP_STACK + \
                                      PRINTER_STACK + \
                                      WINCH_STACK + FLOP_STACK + \
                                      MEM_STACK + CLOCK_STACK + SYS_STACK)
/*
** some notes about the following table:
**  1) The tty_task should always be first so that other tasks can use printf
**     if their initialisation has problems.
**  2) If you add a new kernel task, add it after the amoeba_tasks and before
**     the printer task.
**  3) The task name is used for process status (F1 key) and must be six (6)
**     characters in length.  Pad it with blanks if it is too short.
*/

PUBLIC struct tasktab tasktab[] = {
           tty_task,          TTY_STACK,       "TTY   ",
#ifdef AM_KERNEL
           amint_task,                 AMINT_STACK,    "AMINT ",
           amoeba_task,                AMOEBA_STACK,  "AMTASK",
           amoeba_task,                AMOEBA_STACK,  "AMTASK",
           amoeba_task,                AMOEBA_STACK,  "AMTASK",
           amoeba_task,                AMOEBA_STACK,  "AMTASK",
#endif
           swap_task,         SWAP_STACK,      "SWAPER",
           printer_task,               PRINTER_STACK,  "PRINTR",
           winchester_task,   WINCH_STACK,     "WINCHE",
           floppy_task,                FLOP_STACK,     "FLOPPY",
           mem_task,          MEM_STACK,       "RAMDSK",
           clock_task,                 CLOCK_STACK,    "CLOCK ",
           sys_task,          SYS_STACK,       "SYS   ",
           0,                          0,              "IDLE  ",
           0,                          0,              "MM    ",
           0,                          0,              "FS    ",
           0,                          0,              "INIT  "
```

```
};

int t_stack[TOT_STACK_SPACE/sizeof (int)];

int k_stack[K_STACK_BYTES/sizeof (int)];            /* The kernel stack. */


/*
** The number of kernel tasks must be the same as NR_TASKS.
** If NR_TASKS is not correct then you will get the compile error:
**   multiple case entry for value 0
** The function __dummy is never called.
*/

#define NKT (sizeof tasktab / sizeof (struct tasktab) - (INIT_PROC_NR + 1))
__dummy()
{
        switch(0)
        {
        case 0:
        case (NR_TASKS == NKT):
                ;
        }
}
```

## APPENDIX B - MODIFICATIONS TO MEMORY MANAGER CODE

# Appendix B-2 - MODIFICATIONS TO MEMORY MANAGER CODE



```
                    const.h

>   #define SWAP_MODE          0777     /* mode to use on swap device files */


                    mproc.h

>   unsigned mp_ssw_map;                /* bitmap of signals recvd while swapped */
>   int mp_deadchild;          /* >0 means WAITING SWAPPED proc's child died*/
>   #define SWAPPED            0100     /* process is swapped out */
>   #define FKSWAPPED          0200     /* process is swapped out by fork */
>   #define WASPWS             0400     /* process was P/W & SWAPPED & then awakened */
```

# Appendix B-3 - MODIFICATIONS TO MEMORY MANAGER CODE



```
  /*===========================================================================*
   *                              alloc_mem                                    *
   *===========================================================================*/
  PUBLIC phys_clicks alloc_mem(clicks)
  phys_clicks clicks;              /* amount of memory requested */
  {
  /* Allocate a block of memory from the free bist using first fit. The block
   * consists of a sequence of contiguous bytes, whose length in clicks is
   * given by 'clicks'. A pointer to the block is retumed. The block is
   * always on a click boundary. This procedure is called when memory is
   * needed for FORK or EXEC.
   */

   register struct hole *hp, *prev_ptr;
   phys_clicks old_base;

>  while(TRUE) {
>   hp = hole_head;
>   while (hp != NIL_HOLE) {
>         if (hp->h_len >= clicks) {
>                 /* We found a hole that is big enough. Use it. */
>                 old_base = hp->h_base;        /* remember where it stsrted */
>                 hp->h_base += clicks;         /* bite a piece off */
>                 hp->h_len -= clicks; /* ditto */
>
>                 /* If hole is only partly used, reduce size and retum. */
>                 if (hp->h_len != 0) retum(old_base);
>
>                 /* The entire hole has been used up. Manipulate free list. */
>                 del_slot(prev_ptr, hp);
>                 retum(old_base);
>         }
>
>         prev_ptr = hp;
>         hp = hp->h_next;
>   }
>   if(tot_hole() < clicks)
>         break;
>   compact();        /* mem is available, compact to get it */
>  }
>  retum(NO_MEM);
> }


  /*===========================================================================*
   *                              tot_hole                                     *
   *===========================================================================*/
> PUBLIC phys_clicks tot_bole()
> {
> /* Scan the hole list and retum sum of all holes. */
>
>  register struct hole *hp;
>  register phys_clicks total;
```

```
>
>    hp = hole_head;
>    total = 0;
>    do {
>            total += hp->h_len;
>    }while ((hp = hp->h_next) != NIL_HOLE);
>    return(total);
>  }
>

   /*===========================================================================*
    *                              compact                                      *
    *===========================================================================*/
>  PUBLIC compact()
>  {
>  /* Go through the memory hole map and eliminate all holes except one
>   * by actually moving process images into the empty spaces.
>   */
>
>    phys_clicks hole_base, old_base, proc_size;
>    long hb, ob, ps;
>    struct mproc *rmp;
>    unsigned old_state;
>    int found;
>
>    sys_lock(LOCK, &old_state);
>    while(hole_head->h_next != NIL_HOLE) {
>      /* find first hole */
>      hole_base = hole_head->h_base;
>
>      /* find proc just above this hole */
>      found = 0;
>      for(rmp = &mproc[INIT_PROC_NR +1]; rmp < &mproc[NR_PROCS]; rmp++) {
>        if( ((rmp->mp_flags & IN_USE) == 0)        ||
>            (rmp->mp_flags & (SWAPPED | FKSWAPPED)) ||
>            (rmp->mp_flags & HANGING) )
>                    continue;
>        if( (rmp->mp_seg[T].mem_phys - hole_base) == hole_head->h_len) {
>          found = 1;
>          break;
>        }
>      }
>      if(found == 0) {
>        return;
>      }
>      old_base = rmp->mp_seg[T].mem_phys;
>      proc_size = rmp->mp_seg[S].mem_phys + rmp->mp_seg[S].mem_len - old_base;
>
>      hb = (long) hole_base << CLICK_SHIFT;
>      ob = (long) old_base  << CLICK_SHIFT;
>      ps = (long) proc_size << CLICK_SHIFT;
>
>      if(mem_copy(ABS,0,ob, ABS,0,hb, ps) == OK) {
```

# Appendix B-5 - MODIFICATIONS TO MEMORY MANAGER CODE



```
        /* free old core */
        free_mem(old_base, proc_size);
        /* allocate new core */
        if(hole_base != alloc_mem(proc_size)) {
          sys_lock(RESTORE, &old_state);
          panic("compact - alloc error",1);
        }

        /* setup process table map */
        rmp->mp_seg[S].mem_phys = rmp->mp_seg[S].mem_phys -
                            rmp->mp_seg[T].mem_phys + hole_base;
        rmp->mp_seg[D].mem_phys = rmp->mp_seg[T].mem_len  + hole_base;
        rmp->mp_seg[T].mem_phys = hole_base;
        /* tell kernel */
        sys_newmap( (int) (rmp - mproc), rmp->mp_seg, FALSE);
      }

    }
    sys_lock(RESTORE, &old_state);
  }
```

# Appendix B-6 - MODIFICATIONS TO MEMORY MANAGER CODE



```
/*===========================================================================*
 *                              do_exec                                      *
 *===========================================================================*/
PUBLIC int do_exec()
{
/* Perform the execve(name, argv, envp) call.  The user library builds a
 * complete stack image, including pointers, args, environ, etc.  The stack
 * is copied to a buffer inside MM, and then to the new core image.
 */

  char mbuf[MAX_ISTACK_BYTES];       /* buffer for stack and zeroes */
  char swap_name[4];
  char *new_sp, *a, *psrc, *pdst;
  int s, r, in_fd, out_fd, swap_fd, ft, sd;
  int swapout = FALSE;
  unsigned loadbytes;
  vir_bytes src, dst, text_bytes, data_bytes, bss_bytes, stk_bytes, vsp;
  phys_bytes tot_bytes;              /* total space for program, including gap */
  phys_clicks old_clk, dbuf_len;
  long sym_bytes, tbytes, xtrabytes;
  vir_clicks sc;
  struct mproc *rmp, *tmpmp;
  struct stat s_buf, d_buf;
  struct mem_map tmm[NR_SEGS];
  union u {
        char name_buf[MAX_PATH];   /* the name of the file to exec */
        char zb[ZEROBUF_SIZE];     /* used to zero bss */
  } u;

  /* Do some validity checks. */
  rmp = mp;
  stk_bytes = (vir_bytes) stack_bytes;
  if (stk_bytes > MAX_ISTACK_BYTES) return(ENOMEM);    /* stack too big */
  if (exec_len <= 0 || exec_len > MAX_PATH) return(EINVAL);

  /* Get the exec file name and see if the file is executable. */
  src = (vir_bytes) exec_name;
  dst = (vir_bytes) u.name_buf;
  r = mem_copy(who, D, (long) src, MM_PROC_NR, D, (long) dst, (long) exec_len);
  if (r != OK) return(r);          /* file name not in user data segment */
  tell_fs(CHDIR, who, 0, 0);       /* temporarily switch to user's directory */
  in_fd = allowed(u.name_buf, &s_buf, X_BIT);      /* is file executable? */
  tell_fs(CHDIR, 0, 1, 0);         /* switch back to MM's own directory */
  if (in_fd < 0) return(in_fd);    /* file was not executable */

  /* Read the file header and extract the segment sizes. */
  sc = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  if (read_header(in_fd, &ft, &text_bytes, &data_bytes, &bss_bytes,
                                &tot_bytes, &sym_bytes, sc)  < 0) {
        close(in_fd);                   /* something wrong with header */
        return(ENOEXEC);
  }
```

```
	/* Fetch the stack from the user before destroying the old core image. */
	src = (vir_bytes) stack_ptr;
	dst = (vir_bytes) mbuf;
	if (mem_copy(who, D, (long) src, MM_PROC_NR, D, (long) dst,
					(long) stk_bytes) != OK) {
		close(in_fd);			/* can't fetch stack (e.g. bad virtual addr) */
		return(EACCES);
	}

	/* Allocate new memory and release old memory.  Fix map and tell kernel. */
>	r = new_mem(text_bytes, data_bytes, bss_bytes, stk_bytes, tot_bytes,
>						u.zb, ZEROBUF_SIZE, tmm, &old_clk);
>	if (r == EXSWAPD) {
>		swapout = TRUE;
>	} else if (r != OK) {
>		close(in_fd);			/* insufficient core or program too big */
>		return(r);
>	}

>	if(! swapout) {
		/* Patch up stack and copy it from MM to new core image. */
		vsp = (vir_bytes) (rmp->mp_seg[S].mem_vir << CLICK_SHIFT);
		vsp += (vir_bytes) (rmp->mp_seg[S].mem_len << CLICK_SHIFT);
		vsp -= stk_bytes;
		patch_ptr(mbuf, vsp);
		src = (vir_bytes) mbuf;
		r = mem_copy(MM_PROC_NR, D, (long) src, who, D, (long) vsp,
				(long) stk_bytes);
		if (r != OK) panic("do_exec stack copy err", NO_NUM);

		/* Read in text and data segments. */
		load_seg(who, in_fd, T, text_bytes);
		load_seg(who, in_fd, D, data_bytes);
#ifdef ATARI_ST
		if (lseek(in_fd, sym_bytes, 1) < 0)
			;		/* error */
		if (relocate(in_fd, mbuf) < 0)
			;		/* error */
#endif
>	} else {
>		/* read T & D from a.out file and write to swap device */
>		/* create swap file */
>		/* change to swap directory */
>		tell_fs(CHDIR, 0, 2, 0);
>		tmpmp = mp;
>		mp = &mproc[MM_PROC_NR];
>
>		/* get swap file name */
>		sd = who;
>		a = swap_name;
>		while(sd) {
>		   *a++ = (sd % 10) + 060;
>		   sd /= 10;
```

```
        }
        *a = 0;
        out_fd = allowed(swap_name, &s_buf, W_BIT);         /* swap_file wrtble? */
        s = allowed(".", &d_buf, W_BIT);
        mp = tmpmp;
        if (out_fd >= 0) close(out_fd);
        if (s >= 0) close(s);
        if (s >= 0 && (out_fd >= 0 || out_fd == ENOENT)) {
           /* File is writable or doesn't exist & dir is writable */
           out_fd = creat(swap_name, SWAP_MODE);
        } else {
           tell_fs(CHDIR, 0, 1, 0);         /* go back to MM's own dir */
           return(ERROR);
        }
        tell_fs(CHDIR, 0, 1, 0);            /* go back to MM's own dir */
        if (out_fd < 0) return(ERROR);

        /* change existing core to a data buffer */
        /* get max available size */
        dbuf_len = MIN(2047, old_clk);

        /* setup data buffer */
        rmp->mp_seg[D].mem_phys = rmp->mp_seg[T].mem_phys;
        rmp->mp_seg[D].mem_len = dbuf_len;

        /* tell kernel about the buffer */
        sys_newmap(who, rmp->mp_seg, TRUE);

        swap_fd = (who << 8) | (D << 6) | out_fd;
        for(r=0; r<2; r++) {
           ttbytes = r ? (long)data_bytes : (long)text_bytes;
           xtrabytes = (long)
                        ((r ? tmm[D].mem_len : tmm[T].mem_len) <<CLICK_SHIFT)
                        - ttbytes;
           while(ttbytes) {
                loadbytes = (unsigned)(MIN( (long)ttbytes,
                                        (long)(dbuf_len << CLICK_SHIFT) ));
                /* read from a.out file */
                load_seg(who, in_fd, D, loadbytes);
                /* write it to swap device */
                a = (char *)(rmp->mp_seg[D].mem_vir << CLICK_SHIFT);

                if(write(swap_fd, a, loadbytes) != loadbytes) {
                   close(in_fd);
                   close(out_fd);
                   panic("do_exec swap device write err", NO_NUM);
                }
                ttbytes -= (long)loadbytes;
           }
           if(xtrabytes)
                /* the number of bytes in the swap file for each      */
             /* segment must be equal to the length of the segment */
                if(lseek(out_fd, xtrabytes, 1) < 0)
```

```
>                               printf("lseek error0");
>               }
>
>               /* write stack to swap device */
>               /* first fix mproc memory map */
>               psrc = (char *)tmm;
>               pdst = (char *)rmp->mp_seg;
>               while(psrc != ((char *)tmm + sizeof(tmm)) )
>                               *(pdst++) = *(psrc++);
>               vsp = (vir_bytes) (rmp->mp_seg[S].mem_vir << CLICK_SHIFT);
>               vsp += (vir_bytes) (rmp->mp_seg[S].mem_len << CLICK_SHIFT);
>               vsp -= stk_bytes;
>               patch_ptr(mbuf, vsp);
>               a = mbuf;
>               if(lseek(out_fd, (long)(vsp -(tmm[S].mem_vir << CLICK_SHIFT)),1)
>                   < (long)0)
>                   printf("lseek error0");
>               if(write(out_fd, a, (unsigned)stk_bytes) != stk_bytes) {
>                       close(in_fd);
>                       close(out_fd);
>                       panic("do_exec swap device write err", NO_NUM);
>               }
>               close(out_fd);
>               free_mem(rmp->mp_seg[T].mem_phys, old_clk);     /* free the memory */
>               rmp->mp_flags |= SWAPPED;  /* mark mproc as  swapped */
>       }

        close(in_fd);                           /* don't need exec file any more */

        /* Take care of setuid/setgid bits. */
        if (s_buf.st_mode & I_SET_UID_BIT) {
                rmp->mp_effuid = s_buf.st_uid;
                tell_fs(SETUID, who, (int) rmp->mp_realuid, (int) rmp->mp_effuid);
        }
        if (s_buf.st_mode & I_SET_GID_BIT) {
                rmp->mp_effgid = s_buf.st_gid;
                tell_fs(SETGID, who, (int) rmp->mp_realgid, (int) rmp->mp_effgid);
        }

        /* Fix up some 'mproc' fields and tell kernel that exec is done. */
>       rmp->mp_deadchild = 0;          /* reset swap wait */
>       rmp->mp_ssw_map = 0;                    /* reset all swap signals */
        rmp->mp_catch = 0;              /* reset all caught signals */
        rmp->mp_flags &=  SEPARATE;             /* turn off SEPARATE bit */
        rmp->mp_flags |= ft;                    /* turn it on for separate I & D files */

        new_sp = (char *) vsp;
>       sys_exec(who, new_sp, swapout);
        return(OK);
}


/*===========================================================================*
```

## Appendix B-10 - MODIFICATIONS TO MEMORY MANAGER CODE



```
 *                              new_mem                               *
 *====================================================================*/
PRIVATE int new_mem(text_bytes, data_bytes, bss_bytes, stk_bytes,
                tot_bytes, bf, zs, tmm, old_clk)
vir_bytes text_bytes;           /* text segment size in bytes */
vir_bytes data_bytes;           /* size of initialized data in bytes */
vir_bytes bss_bytes;            /* size of bss in bytes */
vir_bytes stk_bytes;            /* size of initial stack segment in bytes */
phys_bytes tot_bytes;           /* total memory to allocate, including gap */
char bf[ZEROBUF_SIZE];                  /* buffer to use for zeroing data segment */
int zs;                                 /* true size of 'bf' */
struct mem_map tmm[];                   /* temporary memory map */
phys_clicks *old_clk;           /* # of clicks in old process */
{
/* Allocate new memory and release the old memory. Change the map and report
 * the new map to the kernel. Zero the new core image's bss, gap and stack.
 */

  register struct mproc *rmp;
  vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks, tot_clicks;
  phys_clicks new_base;
  extern phys_clicks alloc_mem();
  extern phys_clicks tot_hole();
#ifdef ATARI_ST
  phys_clicks base, size;
#else
  char *rzp;
  vir_bytes vzb;
  phys_clicks old_clicks;
  phys_bytes bytes, base, count, bss_offset;
#endif

  /* Acquire the new memory. Each of the 4 parts: text, (data+bss), gap,
   * and stack occupies an integral number of clicks, starting at click
   * boundary. The data and bss parts are run together with no space.
   */

  text_clicks = (text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
  gap_clicks = tot_clicks - data_clicks - stack_clicks;
  if ( (int) gap_clicks < 0) return(ENOMEM);

  rmp = mp;
#ifndef ATARI_ST
  old_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
  old_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  if (rmp->mp_flags & SEPARATE) old_clicks += rmp->mp_seg[T].mem_len;
#endif

  /* Check to see if there is a hole big enough. If so, we can risk first
   * releasing the old core image before allocating the new one, since we
```

```
       * know it will succeed.  If there is not enough, return failure.
       */
>     if( (text_clicks + tot_clicks) > (tot_hole() + old_clicks) ) {
>       /* core is not available, must swap proc out, first get new map */
>       tmm[T].mem_len = text_clicks;
>       tmm[T].mem_phys = rmp->mp_seg[T].mem_phys;
>       tmm[D].mem_len = data_clicks;
>       tmm[D].mem_phys = tmm[T].mem_phys + text_clicks;
>       tmm[S].mem_len = stack_clicks;
>       tmm[S].mem_phys = tmm[D].mem_phys + data_clicks + gap_clicks;
>       tmm[T].mem_vir = 0;
>       tmm[D].mem_vir = 0;
>       tmm[S].mem_vir = tmm[D].mem_vir + data_clicks + gap_clicks;
>       *old_clk = old_clicks;
>       return(EXSWAPD);
>     }

    #ifndef ATARI_ST
      /* There is enough memory for the new core image.  Release the old one. */
      free_mem(rmp->mp_seg[T].mem_phys, old_clicks);  /* free the memory */
    #endif

      /* We have now passed the point of no return.  The old core image has been
       * forever lost.  The call must go through now.  Set up and report new map.
       */
      new_base = alloc_mem(text_clicks + tot_clicks);      /* new core image */
      if (new_base == NO_MEM) panic("MM hole list is inconsistent", NO_NUM);
      rmp->mp_seg[T].mem_len = text_clicks;
      rmp->mp_seg[T].mem_phys = new_base;
      rmp->mp_seg[D].mem_len = data_clicks;
      rmp->mp_seg[D].mem_phys = new_base + text_clicks;
      rmp->mp_seg[S].mem_len = stack_clicks;
      rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys + data_clicks + gap_clicks;
    #ifdef ATARI_ST
      rmp->mp_seg[T].mem_vir = rmp->mp_seg[T].mem_phys;
      rmp->mp_seg[D].mem_vir = rmp->mp_seg[D].mem_phys;
      rmp->mp_seg[S].mem_vir = rmp->mp_seg[S].mem_phys;
    #else
      rmp->mp_seg[T].mem_vir = 0;
      rmp->mp_seg[D].mem_vir = 0;
      rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir + data_clicks + gap_clicks;
    #endif
    #ifdef ATARI_ST
      sys_fresh(who, rmp->mp_seg, (phys_clicks)(data_bytes >> CLICK_SHIFT),
                                    &base, &size);
      free_mem(base, size);
    #else
      sys_newmap(who, rmp->mp_seg, FALSE);             /* report new map to the kernel */

      /* Zero the bss, gap, and stack segment. Start just above text.  */
      for (rzp = &bf[0]; rzp < &bf[zs]; rzp++) *rzp = 0;    /* clear buffer */
      bytes = (phys_bytes) (data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
      vzb = (vir_bytes) bf;
```

## Appendix B-12 - MODIFICATIONS TO MEMORY MANAGER CODE



```
  base = (long) rmp->mp_seg[T].mem_phys + rmp->mp_seg[T].mem_len;
  base = base << CLICK_SHIFT;
  bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
  base += bss_offset;
  bytes -= bss_offset;

  while (bytes > 0) {
	count = (long) MIN(bytes, (phys_bytes) zs);
	if (mem_copy(MM_PROC_NR, D, (long) vzb, ABS, 0, base, count) != OK)
		panic("new_mem can't zero", NO_NUM);
	base += count;
	bytes -= count;
  }
#endif
  return(OK);
}


/*===========================================================================*
 *				load_seg				     *
 *===========================================================================*/
PUBLIC load_seg(usr, fd, seg, seg_bytes)
int usr;			/* user slot in proc table */
int fd;				/* file descriptor to read from */
int seg;			/* T or D */
vir_bytes seg_bytes;		/* how big is the segment */
{
/* Read in text or data from the exec file and copy to the new core image.
 * This procedure is a little bit tricky.  The logical way to load a segment
 * would be to read it block by block and copy each block to the user space
 * one at a time.  This is too slow, so we do something dirty here, namely
 * send the user space and virtual address to the file system in the upper
 * 10 bits of the file descriptor, and pass it the user virtual address
 * instead of a MM address.  The file system copies the whole segment
 * directly to user space, bypassing MM completely.
 */

  int new_fd, bytes;
  char *ubuf_ptr;
  struct mproc *rmp;

  new_fd = (usr << 8) | (seg << 6) | fd;
  rmp = &mproc[usr];
  ubuf_ptr = (char *) ((vir_bytes)rmp->mp_seg[seg].mem_vir << CLICK_SHIFT);
  while (seg_bytes) {
	bytes = 31*1024;		/* <= 32767 */
	if (seg_bytes < bytes)
		bytes = (int)seg_bytes;
	if (read(new_fd, ubuf_ptr, bytes) != bytes) {
		panic("loadseg read err", NO_NUM);
		break;			/* error */
	}
	ubuf_ptr += bytes;
```

```
                seg_bytes -= bytes;
        }
    }
```

# Appendix B-14 - MODIFICATIONS TO MEMORY MANAGER CODE



```
  /*=====================================================================*
   *                             do_fork                        *
   *=====================================================================*/
  PUBLIC int do_fork()
  {
  /* The process pointed to by 'mp' has forked.  Create a child process. */

    register struct mproc *rmp;     /* pointer to parent */
    register struct mproc *rmc;     /* pointer to child */
    int i, child_nr, t;
    char *sptr, *dptr;
    phys_clicks prog_clicks, child_base;
    extern phys_clicks alloc_mem();
    extern phys_clicks tot_hole();
    int swapout = FALSE;
    vir_bytes new_sp;
  #ifndef ATARI_ST
    long prog_bytes;
    long parent_abs, child_abs;
  #endif

    /* If tables might fill up during FORK, don't even start since recovery half
     * way through is such a nuisance.
     */

    rmp = mp;
    if (procs_in_use == NR_PROCS) return(EAGAIN);
    if (procs_in_use >= NR_PROCS - LAST_FEW && rmp->mp_effuid != 0)return(EAGAIN);

    /* Determine how much memory to allocate. */
    prog_clicks = (phys_clicks) rmp->mp_seg[S].mem_len;
    prog_clicks += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  #ifndef ATARI_ST
    if (rmp->mp_flags & SEPARATE) prog_clicks += rmp->mp_seg[T].mem_len;
    prog_bytes = (long) prog_clicks << CLICK_SHIFT;
  #endif

>   if ( (prog_clicks > tot_hole()) ||
>       ( (child_base = alloc_mem(prog_clicks)) == NO_MEM) ) {
>             swapout = TRUE;
>             /* adjust parents memory map, if necessary */
>             sys_getsp(who, &new_sp);
>             if(adjust(rmp, (vir_clicks) rmp->mp_seg[D].mem_len, new_sp) != OK)
>                     return(EAGAIN);
>   }

  #ifndef ATARI_ST
>   if(! swapout) {
            /* Create a copy of the parent's core image for the child. */
            child_abs = (long) child_base << CLICK_SHIFT;
            parent_abs = (long) rmp->mp_seg[T].mem_phys << CLICK_SHIFT;
            i = mem_copy(ABS, 0, parent_abs, ABS, 0, child_abs, prog_bytes);
            if ( i < 0) panic("do_fork can't copy", i);
```

```
    }
  #endif

    /* Find a slot in 'mproc' for the child process.  A slot must exist. */
    for (rmc = &mproc[0]; rmc < &mproc[NR_PROCS]; rmc++)
          if ( (rmc->mp_flags & IN_USE) == 0) break;

    /* Set up the child and its memory map; copy its 'mproc' slot from parent. */
    child_nr = (int)(rmc - mproc);  /* slot number of the child */
    sptr = (char *) rmp;            /* pointer to parent's 'mproc' slot */
    dptr = (char *) rmc;            /* pointer to child's 'mproc' slot */
    i = sizeof(struct mproc);       /* number of bytes in a proc slot. */
    while (i--) *dptr++ = *sptr++;/* copy from parent slot to child's */

    rmc->mp_parent = who;                    /* record child's parent */
  #ifndef ATARI_ST
>   if(! swapout) {
          rmc->mp_seg[T].mem_phys = child_base;
          rmc->mp_seg[D].mem_phys = child_base + rmc->mp_seg[T].mem_len;
          rmc->mp_seg[S].mem_phys = rmc->mp_seg[D].mem_phys +
                         (rmp->mp_seg[S].mem_phys - rmp->mp_seg[D].mem_phys);
>   } else {
>         /* swapout parent's image for child, don't free parent's core */
>         if( swap_out(child_nr, rmc, rmp, FALSE) != OK) {
>               return(EAGAIN);
>         } else {
>               rmc->mp_flags |= FKSWAPPED;
>         }
>   }
  #endif
    rmc->mp_exitstatus = 0;
    rmc->mp_sigstatus = 0;
>   rmc->mp_deadchild = 0;        /* reset swap wait */
>   rmc->mp_ssw_map = 0;                     /* reset all swap signals */
    procs_in_use++;

    /* Find a free pid for the child and put it in the table. */
    do {
          t = 0;                             /* 't' = 0 means pid still free */
          next_pid = (next_pid < 30000 ? next_pid + 1 : INIT_PROC_NR + 1);
          for (rmp = &mproc[0]; rmp < &mproc[NR_PROCS]; rmp++)
                if (rmp->mp_pid == next_pid || rmp->mp_procgrp == next_pid) {
                      t = 1;
                      break;
                }
          rmc->mp_pid = next_pid;       /* assign pid to child */
    } while (t);

    /* Set process group. */
    if (who == INIT_PROC_NR) rmc->mp_procgrp = rmc->mp_pid;

    /* Tell kernel and file system about the (now successful) FORK. */
  #ifdef ATARI_ST
```

# Appendix B-16 - MODIFICATIONS TO MEMORY MANAGER CODE

Jul 6 14:41 1989 FORKEXIT.C Page 3

```
    sys_fork(who, child_nr, rmc->mp_pid, child_base);
  #else
>   sys_fork(who, child_nr, rmc->mp_pid, swapout);
  #endif

    tell_fs(FORK, who, child_nr, rmc->mp_pid);

  #ifndef ATARI_ST
    /* Report child's memory map to kernel. */
    if(! swapout) {
      sys_newmap(child_nr, rmc->mp_seg, FALSE);
    }
  #endif

>   if(! swapout) {
          /* Reply to child to wake it up. */
          reply(child_nr, 0, 0, NIL_PTR);
>   }
    return(next_pid);                 /* child's pid */
  }


  /*===========================================================================*
   *                              do_mm_exit                                   *
   *===========================================================================*/
  PUBLIC int do_mm_exit()
  {
  /* Perform the exit(status) system call. The real work is done by mm_exit(),
   * which is also called when a process is killed by a signal.
   */

    mm_exit(mp, status);
    dont_reply = TRUE;                /* don't reply to newly terminated process */
    return(OK);                       /* pro forma return code */
  }


  /*===========================================================================*
   *                              mm_exit                                      *
   *===========================================================================*/
  PUBLIC mm_exit(rmp, exit_status)
  register struct mproc *rmp;         /* pointer to the process to be terminated */
  int exit_status;                    /* the process' exit status (for parent) */
  {
  /* A process is done.  If parent is waiting for it, clean it up, else hang. */
  #ifdef ATARI_ST
    phys_clicks base, size;
  #endif
    phys_clicks s;
    register int proc_nr = (int)(rmp - mproc);

    /* How to terminate a process is determined by whether or not the
     * parent process has already done a WAIT.  Test to see if it has.
```

```
  */
  rmp->mp_exitstatus = (char) exit_status;  /* store status in 'mproc' */

  if (mproc[rmp->mp_parent].mp_flags & WAITING) {
        if(mproc[rmp->mp_parent].mp_flags & (SWAPPED | FKSWAPPED)) {
                /* parent is swapped & waiting, delay cleanup by falsely */
                /* marking child HANGING, setting deadchild, and telling */
                /* SWAP_TASK */
                rmp->mp_flags |= HANGING;
                mproc[rmp->mp_parent].mp_deadchild = proc_nr;
                mproc[rmp->mp_parent].mp_flags &=  WAITING;
                reply(SWAP_TASK, CORE_IS_NEEDED,
                    (int)(&mproc[rmp->mp_parent] - mproc), NIL_PTR);
        } else
                cleanup(rmp);                   /* release parent and tell everybody */
  } else
        rmp->mp_flags |= HANGING;  /* Parent not waiting.  Suspend proc */

  /* If the exited process has a timer pending, kill it. */
  if (rmp->mp_flags & ALARM_ON) set_alarm(proc_nr, (unsigned) 0);

#ifdef AM_KERNEL
/* see if an amoeba transaction was pending or a putrep needed to be done */
  am_check_sig(proc_nr, 1);
#endif

  /* Tell the kernel and FS that the process is no longer runnable. */
#ifdef ATARI_ST
  sys_xit(rmp->mp_parent, proc_nr, &base, &size);
  free_mem(base, size);
#else
  sys_xit(rmp->mp_parent, proc_nr);
#endif
  tell_fs(EXIT, proc_nr, 0, 0);  /* file system can free the proc slot */

#ifndef ATARI_ST
  /* Release the memory occupied by the child. */
  s = (phys_clicks) rmp->mp_seg[S].mem_len;
  s += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
  if (rmp->mp_flags & SEPARATE) s += rmp->mp_seg[T].mem_len;
  free_mem(rmp->mp_seg[T].mem_phys, s);          /* free the memory */
#endif

}


/*===========================================================================*
 *                              do_wait                                      *
 *===========================================================================*/
PUBLIC int do_wait()
{
/* A process wants to wait for a child to terminate. If one is already waiting,
 * go clean it up and let this WAIT call terminate.  Otherwise, really wait.
```

```
 */

 register struct mproc *rp;
 register int children;

 /* A process calling WAIT never gets a reply in the usual way via the
  * reply() in the main loop.  If a child has already exited, the routine
  * cleanup() sends the reply to awaken the caller.
  */

 /* Is there a child waiting to be collected? */
 children = 0;
 for (rp = &mproc[0]; rp < &mproc[NR_PROCS]; rp++) {
         if ( (rp->mp_flags & IN_USE) && rp->mp_parent == who) {
                 children++;
                 if (rp->mp_flags & HANGING) {
                         cleanup(rp);            /* a child has already exited */
                         dont_reply = TRUE;
                         return(OK);
                 }
         }
 }

 /* No child has exited.  Wait for one, unless none exists. */
 if (children > 0) {                 /* does this process have any children? */
         mp->mp_flags |= WAITING;
         dont_reply = TRUE;

         for (rp = &mproc[INIT_PROC_NR + 1]; rp < &mproc[NR_PROCS]; rp++ ) {
                 if ( (rp->mp_flags & IN_USE) == 0) continue;
                 if( (rp->mp_flags & (SWAPPED | FKSWAPPED)) &&
                 (rp->mp_flags & (PAUSED | WAITING) == 0) ) {
                         ms.m_source = MM_PROC_NR;
                         ms.m_type = CORE_IS_FREE;
                         send(SWAP_TASK, &ms);
                         break;
                 }
         }
         return(OK);                 /* yes - wait for one to exit */
 } else
         return(ECHILD);             /* no - parent has no children */
}
```

# Appendix B-19 - MODIFICATIONS TO MEMORY MANAGER CODE

Jul 6 14:41 1989 MAIN.C Page 1

```
/*===========================================================================*
 *                              main                                         *
 *===========================================================================*/
PUBLIC main()
{
/* Main routine of the memory manager. */

  int error;

  mm_init();                        /* initialize memory manager tables */

  /* This is MM's main loop-  get work and do it, forever and forever. */
  while (TRUE) {
        /* Wait for message. */
        get_work();                 /* wait for an MM system call */
        mp = &mproc[who];

        /* Set some flags. */
        error = OK;
        dont_reply = FALSE;
        err_code = -999;

        /* If the call number is valid, perform the call. */
        if (mm_call < 0 || mm_call >= NCALLS)
                error = E_BAD_CALL;
        else
                error = (*call_vec[mm_call])(0);

        /* Send the results back to the user to indicate completion. */
        if (dont_reply) continue;       /* no reply for EXIT and WAIT */
        if (mm_call == EXEC && error == OK) continue;
        reply(who, error, result2, res_ptr);
  }
}



/*===========================================================================*
 *                              reply                                        *
 *===========================================================================*/
PUBLIC reply(proc_nr, result, res2, respt)
int proc_nr;                    /* process to reply to */
int result;             /* result of the call (usually OK or error #)*/
int res2;               /* secondary result */
char *respt;                    /* result if pointer */
{
/* Send a reply to a user process. */

  register struct mproc *proc_ptr;

  /* To make MM robust, check to see if destination is still alive. */
  if(proc_nr != SWAP_TASK) {
    proc_ptr = &mproc[proc_nr];
```

# Appendix B-20 - MODIFICATIONS TO MEMORY MANAGER CODE



```
      if ( (proc_ptr->mp_flags&IN_USE) == 0 || (proc_ptr->mp_flags&HANGING))
        return;
    }
    reply_type = result;
    reply_i1 = res2;
    reply_p1 = respt;
    if (send(proc_nr, &mm_out) != OK) panic("MM can't reply", NO_NUM);
  }
```

# Appendix B-21 - MODIFICATIONS TO MEMORY MANAGER CODE



```
#include "../h/const.h"
#include "../h/type.h"
#include "../h/callnr.h"
#include "../h/com.h"
#include "../h/error.h"
#include "../h/stat.h"
#include "../h/signal.h"
#include "const.h"
#include "glo.h"
#include "mproc.h"
#include "param.h"

#define LONG1            m2_l1     /* message slot to carry long bitmap of */
                                   /* pause/wait procs, if proc table has  */
                                   /* more than 32 slots, another method is*/
                                   /* needed */
PRIVATE struct mproc *rmp;

/*===========================================================================*
 *                              do_swout                                     *
 *===========================================================================*/
do_swout(num)
int num;
{
/* perform request from SWAP_TASK to do swapout of a particular process
 */
int pnum;

        mp = &mproc[MM_PROC_NR];          /* mp points to MM */
        if(num)
                pnum = num;
        else
                pnum = mm_in.PROC1;
        if( (pnum < 0) || (pnum >NR_PROCS) ) {
                printf("DSO1: swapout proc out of range: %d0,pnum);
                return(ERROR);
        }

        rmp = &mproc[pnum];          /* rmp points to swapout proc */

        if(swap_out(pnum, rmp, rmp, TRUE) == OK) {
                rmp->mp_flags |= SWAPPED;
                dont_reply= TRUE;
                mm_out.m_type = SWAP_OUT_COMPL;
                mm_out.PROC1 = pnum;
                if(rmp->mp_flags & (PAUSED | WAITING))
                        mm_out.PROC2 = TRUE;
                else
                        mm_out.PROC2 = FALSE;
                if (send(SWAP_TASK,&mm_out) != OK)
                        panic("mswap can't send mes", NO_NUM);
                return(SWAP_OUT_COMPL);
        } else {
```

Jul 6 14:41 1989 MSWAP.C Page 2

```
                        result2 = pnum;
                        return(SWAP_OUT_FAILED);
                }
}

/*===========================================================================*
 *                              swap_out                                     *
 *===========================================================================*/
swap_out(swap_proc, rmc, rmp, clear_mem)
int swap_proc;
struct mproc *rmc;      /* child process */
struct mproc *rmp;      /* parent process */
int clear_mem;
{
/* do actual work of swapping out the process image to swap device */
phys_clicks s;
int type;
struct mproc *tmp;

        tmp = mp;
        mp = &mproc[MM_PROC_NR];
        if(clear_mem) type = 1;
        else type = 2;

        if(dump_core(rmc, rmp, type) != OK) {
                printf("SWAPOUT ERROR0);
                mp = tmp;
                return(ERROR);
        }

        mp = tmp;
        if(clear_mem) {
            /* Release the memory occupied by the process. */
            s = (phys_clicks) rmp->mp_seg[S].mem_len;
            s += (rmp->mp_seg[S].mem_vir - rmp->mp_seg[D].mem_vir);
            if (rmp->mp_flags & SEPARATE) s += rmp->mp_seg[T].mem_len;
            free_mem(rmp->mp_seg[T].mem_phys, s);      /* free the memory */
        }
        return(OK);
}

/*===========================================================================*
 *                              do_swin                                      *
 *===========================================================================*/
do_swin()
{
/* perform request from SWAP_TASK to do swapin of a particular process
 */
int i, num;
phys_clicks new_base, tot_clicks;
extern phys_clicks alloc_mem(), tot_hole();
extern cleanup();
struct mproc *rpp;
```

Jul 6 14:41 1989 MSWAP.C Page 3

```
long bitmap;

        num = mm_in.PROC1;
        mp = &mproc[MM_PROC_NR];            /* mp points to MM */
        rmp = &mproc[num]; /* rmp points to swapin process */
        tot_clicks = rmp->mp_seg[S].mem_phys -
         rmp->mp_seg[T].mem_phys +
         rmp->mp_seg[S].mem_len;
        if( (tot_hole() >= tot_clicks) &&
           ((new_base = alloc_mem(tot_clicks)) != NO_MEM) ) {
                rmp->mp_flags &= SWAPPED;

                if(swap_in(num, rmp, new_base) == OK) {
                        if(rmp->mp_flags & FKSWAPPED) {
                                rmp->mp_flags &= FKSWAPPED;
                                /* wake up forked child */
                                reply(num, 0, 0, NIL_PTR);
                        }
                        /* do all processing required due to   */
                        /* proc being swapped out, messy stuff */
                        /* process signals recvd during swapout */
                        if(rmp->mp_ssw_map) {
                                for(i=1; i<=NR_SIGS; i++) {
                                        if(rmp->mp_ssw_map & (1 << i -1))
                                                sig_proc(rmp, i);
                                }
                                rmp->mp_ssw_map = 0;
                        }
                        /* if proc was WAIT & SWAPPED & a child died */
                        if(rmp->mp_deadchild) {
                                cleanup(&mproc[rmp->mp_deadchild]);
                                rmp->mp_deadchild = 0;
                        }
                        /* wake up P/W proc awakened by signal while SWAPPED */
                        if(rmp->mp_flags & WASPWS) {
                                reply(num, EINTR, 0, NIL_PTR);
                                rmp->mp_flags &= WASPWS;
                        }
                        result2 = num;
                        return(SWAP_IN_COMPL);
                } else {
                        panic("SWAPIN ERROR",NO_NUM);
                }
        } else {
                /* no core available, send kernel list of PAUSED & */
                /* WAITING procs and amount of core needed */
                rmp->mp_flags |= SWAPPED;
                bitmap = 0;
                for(rpp = &mproc[INIT_PROC_NR +1];
                                rpp < &mproc[NR_PROCS]; rpp++) {
                        if( ((rpp->mp_flags & IN_USE) == 0) ||
                           (rpp->mp_flags & (HANGING | SWAPPED | FKSWAPPED)) )
                                continue;
```

```
					if(rpp->mp_flags & (PAUSED | WAITING))
						bitmap |= 1<< (int)(rpp - mproc);
				}
				mm_out.m_type = SWAP_IN_FAILED;
				mm_out.PROC1 = (unsigned)(tot_clicks - tot_hole());
				mm_out.LONG1 = bitmap;
				if (sendrec(SWAP_TASK,&mm_out) != OK)
					panic("mswap can't send mes", NO_NUM);
				if(mm_out.m_type != SWAP_OUT_REQ) {
					dont_reply = TRUE;
					return(SWAP_IN_FAILED);
				} else {
					return(do_swout(mm_out.PROC1));
				}
		}
}


/*===========================================================================*
 *				swap_in					     *
 *===========================================================================*/
PUBLIC int swap_in(swap_proc, rmp, new_base)
int swap_proc;
struct mproc *rmp;
phys_clicks new_base;
{
/* do actual work of swapping in the process image from swap device */

  int fd, tmp;
  char name_buf[5];  /* the name of the file to swapin */
  char zb[ZEROBUF_SIZE];	/* used to zero bss */
  struct stat s_buf;
  phys_clicks gap_clicks;
  extern int load_seg();
  char *rzp;
  vir_bytes vzb;
  phys_bytes bytes, base, count;

  /* Do some validity checks. */

  /* Get the swap file name */
  rzp = name_buf;
  tmp = swap_proc;
  while(tmp) {		/* same algorithm used in dump_core to determine swapname */
    *rzp++ = (tmp % 10) + 060;
    tmp /= 10;
  }
  *rzp = 0;

  tell_fs(CHDIR, 0, 2, 0);  /* temporarily switch to swap dir */
  fd = allowed(name_buf, &s_buf, R_BIT); /* is file readable? */
  tell_fs(CHDIR, 0, 1, 0);		/* switch back to MM's own directory */
  if (fd < 0) {
```

```
                printf("SWAP FILE not readable0);
                return(ERROR);      /* file was not readable */
        }

        /* Fix map with new memory and tell kernel. */
        gap_clicks = rmp->mp_seg[S].mem_phys -
                (rmp->mp_seg[D].mem_phys +
                 rmp->mp_seg[D].mem_len);
        rmp->mp_seg[T].mem_phys = new_base;
        rmp->mp_seg[D].mem_phys = new_base + rmp->mp_seg[T].mem_len;
        rmp->mp_seg[S].mem_phys = rmp->mp_seg[D].mem_phys +
                        rmp->mp_seg[D].mem_len  +
                        gap_clicks;
        rmp->mp_seg[T].mem_vir = 0;
        rmp->mp_seg[D].mem_vir = 0;
        rmp->mp_seg[S].mem_vir = rmp->mp_seg[D].mem_vir +
                        rmp->mp_seg[D].mem_len  +
                        gap_clicks;
        sys_newmap(swap_proc, rmp->mp_seg, FALSE);      /* report new map to the kernel */

        /* Zero the gap */
        for (rzp = zb; rzp < &zb[ZEROBUF_SIZE]; rzp++) *rzp = 0;    /* clear buffer */
        bytes = (phys_bytes) gap_clicks << CLICK_SHIFT;
        vzb = (vir_bytes) zb;
        base = (long) rmp->mp_seg[D].mem_phys + rmp->mp_seg[D].mem_len;
        base = base << CLICK_SHIFT;

        while (bytes > 0) {
                count = (long) MIN(bytes, (phys_bytes) ZEROBUF_SIZE);
                if (mem_copy(MM_PROC_NR, D, (long) vzb, ABS, 0, base, count) != OK)
                        panic("new_mem can't zero", NO_NUM);
                base += count;
                bytes -= count;
        }

        /* Read in text, data and stack segments. */
        load_seg(swap_proc, fd, T, (phys_bytes) rmp->mp_seg[T].mem_len << CLICK_SHIFT);
        load_seg(swap_proc, fd, D, (phys_bytes) rmp->mp_seg[D].mem_len << CLICK_SHIFT);
        load_seg(swap_proc, fd, S, (phys_bytes) rmp->mp_seg[S].mem_len << CLICK_SHIFT);
        close(fd);                      /* don't need swap file any more */

        tell_fs(CHDIR, 0, 2, 0);  /* temporarily switch to swap dir */
        if(unlink(name_buf) != 0)
                printf("unlink error: %s0,name_buf);
        tell_fs(CHDIR, 0, 1, 0);        /* switch back to MM's own directory */

        return(OK);
}
```

# Appendix B-26 - MODIFICATIONS TO MEMORY MANAGER CODE



```
/*===========================================================================*
 *                              check_sig                                    *
 *===========================================================================*/
PRIVATE int check_sig(proc_id, sig_nr, send_uid)
int proc_id;                    /* pid of process to signal, or 0 or -1 */
int sig_nr;             /* which signal to send (1-16) */
uid send_uid;                   /* identity of process sending the signal */
{
/* Check to see if it is possible to send a signal. The signal may have to be
 * sent to a group of processes. This routine is invoked by the KILL system
 * call, and also when the kernel catches a DEL or other signal. SIGALRM too.
 */

  register struct mproc *rmp;
  int count, send_sig;
  unshort mask;
  extern unshort core_bits;

  if (sig_nr < 1 || sig_nr > NR_SIGS) return(EINVAL);
  count = 0;                    /* count # of signals sent */
  mask = 1 << (sig_nr - 1);

  /* Search the proc table for processes to signal. Several tests are made:
   *       - if proc's uid != sender's, and sender is not superuser, don't signal
   *       - if specific process requested (i.e., 'procpid' > 0), check for match
   *       - if a process has already exited, it can't receive signals
   *       - if 'proc_id' is 0 signal everyone in same process group except caller
   */
  for (rmp = &mproc[INIT_PROC_NR + 1]; rmp < &mproc[NR_PROCS]; rmp++ ) {
        if ( (rmp->mp_flags & IN_USE) == 0) continue;
        send_sig = TRUE;   /* if it's FALSE at end of loop, don't signal */
        if (send_uid != rmp->mp_effuid && send_uid != SUPER_USER)send_sig=FALSE;
        if (proc_id > 0 && proc_id != rmp->mp_pid) send_sig = FALSE;
        if (rmp->mp_flags & HANGING) send_sig = FALSE;   /*don't wake the dead*/
        if (proc_id == 0 && mp->mp_procgrp != rmp->mp_procgrp) send_sig = FALSE;
        if (send_uid == SUPER_USER && proc_id == -1) send_sig = TRUE;

        /* SIGALARM is a little special. When a process exits, a clock signal
         * can arrive just as the timer is being turned off. Also, turn off
         * ALARM_ON bit when timer goes off to keep it accurate.
         */
        if (sig_nr == SIGALRM) {
                if ( (rmp->mp_flags & ALARM_ON) == 0) continue;
                if (send_sig) rmp->mp_flags &= ALARM_ON;
        }

        if (send_sig == FALSE) continue;
        count++;
        if (rmp->mp_ignore & mask) continue;

#ifdef AM_KERNEL
        /* see if an amoeba transaction should be signalled */
        Tfs = am_check_sig(rmp - mproc, 0);
```

```
  #endif

>               /* Send the signal or kill the process, possibly with core dump. */
>               if(rmp->mp_flags & (SWAPPED | FKSWAPPED))
>                       /* proc is SWAPPED, delay processing until swapped in */
>                       rmp->mp_ssw_map |= 1 << (sig_nr -1);
>               else
>                       sig_proc(rmp, sig_nr);

                /* If process is hanging on PAUSE, WAIT, tty, pipe, etc. release it. */
                unpause((int)(rmp - mproc));    /* check to see if process is paused */
                if (proc_id > 0) break;         /* only one process being signaled */
    }

    /* If the calling process has killed itself, don't reply. */
    if ((mp->mp_flags & IN_USE) == 0 || (mp->mp_flags & HANGING))dont_reply =TRUE;
    return(count > 0 ? OK : ESRCH);
  }


  /*===========================================================================*
   *                              do_pause                                     *
   *===========================================================================*/
  PUBLIC int do_pause()
  {
  /* Perform the pause() system call. */
    struct mproc *rmp;

    mp->mp_flags |= PAUSED;    /* turn on PAUSE bit */
    dont_reply = TRUE;

>   for (rmp = &mproc[INIT_PROC_NR + 1]; rmp < &mproc[NR_PROCS]; rmp++ ) {
>     if ( (rmp->mp_flags & IN_USE) == 0) continue;
>     if( (rmp->mp_flags & (SWAPPED | FKSWAPPED)) &&
>         (rmp->mp_flags & (PAUSED | WAITING) == 0) ) {
>       reply(SWAP_TASK, CORE_IS_FREE, 0, NIL_PTR);
>       break;
>     }
>   }
    return(OK);
  }


  /*===========================================================================*
   *                              unpause                                      *
   *===========================================================================*/
  PRIVATE unpause(pro)
  int pro;                        /* which process number */
  {
  /* A signal is to be sent to a process. If that process is hanging on a
   * system call, the system call must be terminated with EINTR. Possible
   * calls are PAUSE, WAIT, READ and WRITE, the latter two for pipes and ttys.
   * First check if the process is hanging on PAUSE or WAIT. If not, tell FS,
```

```
   * so it can check for READs and WRITEs from pipes, ttys and the like.
   */

    register struct mproc *rmp;

    rmp = &mproc[pro];

    /* Check to see if process is hanging on a PAUSE call. */
    if ( (rmp->mp_flags & PAUSED) && (rmp->mp_flags & HANGING) == 0) {
          rmp->mp_flags &=  PAUSED;   /* turn off PAUSED bit */
>         if(rmp->mp_flags & (SWAPPED | FKSWAPPED)) {
>                 /* PAUSED swapped proc awakened, delay notice & tell kernel */
>                 rmp->mp_flags |= WASPWS;
>                 reply(SWAP_TASK, CORE_IS_NEEDED, pro, NIL_PTR);
>         } else
                  reply(pro, EINTR, 0, NIL_PTR);
          return;
    }

    /* Check to see if process is hanging on a WAIT call. */
    if ( (rmp->mp_flags & WAITING) && (rmp->mp_flags & HANGING) == 0) {
          rmp->mp_flags &=  WAITING; /* turn off WAITING bit */
>         if(rmp->mp_flags & (SWAPPED | FKSWAPPED)) {
>                 /* PAUSED swapped proc awakened, delay notice & tell kernel */
>                 rmp->mp_flags |= WASPWS;
>                 reply(SWAP_TASK, CORE_IS_NEEDED, pro, NIL_PTR);
>         } else
                  reply(pro, EINTR, 0, NIL_PTR);
          return;
    }

#ifdef AM_KERNEL
   /* if it was an amoeba transaction, it is already tidied up by now. */
   if (TTs)
#endif
   /* Process is not hanging on an MM call.  Ask FS to take a look. */
          tell_fs(UNPAUSE, pro, 0, 0);

   return;
 }


/*===========================================================================*
 *                              dump_core                                    *
 *===========================================================================*/
PUBLIC dump_core(rmc, rmp, type)
struct mproc *rmc;    /* child proc for swapout */
struct mproc *rmp;    /* whose core is to be dumped */
int type;             /* 0-dump core; 1-swapout; 2-sp_out w/noadjust */
{
/* Make a core dump on the file "core", if possible. */

>  struct stat s_buf, d_buf;
```

# Appendix B-29 - MODIFICATIONS TO MEMORY MANAGER CODE



```
>     int i, r, s, new_fd, slot, dir, flag, bytes;
>     vir_bytes v_buf, c, new_sp;
>     char *a;
>     struct mproc *xmp;
>     extern char core_name[];
>     extern adjust();
>     char swap_name[4];
>
>     slot = (int)(rmc - mproc);
>     /* Change to working directory of dumpee. */
>     if(type > 0) {
>       dir = 0;/* swapout */
>       flag = 2;
>     } else {
>       dir = slot;          /* dump_core */
>       flag = 0;
>     }
>     tell_fs(CHDIR, dir, flag, 0);
>
>     /* Can file be written? */
>     if( (type == 0) && (rmc->mp_realuid != rmc->mp_effuid) ) {
>             tell_fs(CHDIR, 0, 1, 0);          /* go back to MM's directory */
>             return(ERROR);
>     }
>     if(type == 0) {
>       xmp = mp;                              /* allowed() looks at 'mp' */
>       mp = rmc;
>     }
>
>     if(type > 0) {
>       dir = slot;
>       a = swap_name;
>       while(dir) {
>         *a++ = (dir % 10) + 060;
>         dir /= 10;
>       }
>       *a = 0;
>       r = allowed(swap_name, &s_buf, W_BIT);          /* is swap_file writable */
>     } else
>       r = allowed(core_name, &s_buf, W_BIT);          /* is core_file writable */
>
>     s = allowed(".", &d_buf, W_BIT);          /* is directory writable? */
>
>     if(type == 0)
>       mp = xmp;
>     if (r >= 0) close(r);
>     if (s >= 0) close(s);
>     if ((type > 0) || (rmc->mp_effuid == SUPER_USER))
>        r = 0;  /* su can always dump core */
>
>     if (s >= 0 && (r >= 0 || r == ENOENT)) {
>             /* Either file is writable or it doesn't exist & dir is writable */
>             if(type > 0)
```

```
    r = creat(swap_name, SWAP_MODE);
  else
    r = creat(core_name, CORE_MODE);
  tell_fs(CHDIR, 0, 1, 0);          /* go back to MM's own dir */
  if (r < 0) {
          printf("create error0);
          return(ERROR);
  }

  if(type != 2) {
     /* adjust memory map, if necessary (already done for type 2) */
     sys_getsp(slot, &new_sp);
     if(adjust(rmc, (vir_clicks) rmc->mp_seg[D].mem_len, new_sp)
         != OK) {
          printf("ADJUST ERROR0);
          close(r);
          return(ERROR);
     }
  }

  if(type == 0) {
          rmc->mp_sigstatus |= DUMPED;

          /* First write the memory map of all segments on core file. */
          if (write(r, (char *) rmc->mp_seg, sizeof(rmc->mp_seg)) < 0) {
                  close(r);
                  printf("write memory map error0);
                  return(ERROR);
          }
  }

  if(type == 2)
          slot = (int) (rmp - mproc);
  /* Now loop through segments and write the segments themselves out. */
  for (i = 0; i < NR_SEGS; i++) {
          a = (char *) (rmc->mp_seg[i].mem_vir << CLICK_SHIFT);
          c = (int) (rmc->mp_seg[i].mem_len << CLICK_SHIFT);
          new_fd = ((int)(rmp - mproc) << 8) | (i << 6) | r;

          /* Dump segment. */
          while(c) {
                  bytes = 31 * 1024;
                  if(c < (vir_bytes)bytes)
                          bytes = (int)c;
                  if (write(new_fd, a, bytes) != bytes) {
                          close(r);
                          printf("write segment error0);
                          return(ERROR);
                  }
                  a += bytes;
                  c -= bytes;
          }
  }
```

```
	} else {
		printf("swap file or dir is not writeable0);
		tell_fs(CHDIR, 0, 1, 0);		/* go back to MM's own dir */
		close(r);
		return(ERROR);
	}

	close(r);
	return(OK);
}
```

# Appendix B-32 - MODIFICATIONS TO MEMORY MANAGER CODE



```
/* This file contains the table used to map system call numbers onto the
 * routines that perform them.
 */

#include "../h/const.h"
#include "../h/type.h"
#include "const.h"

#undef EXTERN
#define EXTERN

#include "../h/callnr.h"
#include "glo.h"
#include "mproc.h"
#include "param.h"

/* Miscellaneous */
char core_name[] = {"core"};    /* file name where core images are produced */
char swap_name[] = {"xxxx"};  /* swap device file names */
#ifdef ATARI_ST
/*
 * Creating core files is disabled, except for SIGQUIT and SIGIOT.
 * Set core_bits to 0x0EFC if you want compatibility with UNIX V7.
 */
unshort core_bits = 0x0EFC;     /* which signals cause core images */
#else
unshort core_bits = 0x0EFC;     /* which signals cause core images */
#endif

extern char mm_stack[];
char *stackpt = &mm_stack[MM_STACK_BYTES];  /* initial stack pointer */

extern do_mm_exit(), do_fork(), do_wait(), do_brk(), do_getset(), do_exec();
extern do_signal(), do_kill(), do_pause(), do_alarm();
extern no_sys(), do_ksig(), do_brk2(), do_swin(), do_swout();

#ifdef AM_KERNEL
extern do_amoeba();
#endif

int (*call_vec[NCALLS])() = {
        no_sys,         /*  0 = unused      */
        do_mm_exit,     /*  1 = exit        */
        do_fork,  /*  2 = fork        */
        no_sys,         /*  3 = read        */
        no_sys,         /*  4 = write       */
        no_sys,         /*  5 = open        */
        no_sys,         /*  6 = close       */
        do_wait,  /*  7 = wait        */
        no_sys,         /*  8 = creat       */
        no_sys,         /*  9 = link        */
        no_sys,         /* 10 = unlink      */
        no_sys,         /* 11 = exec        */
```

# Appendix B-33 - MODIFICATIONS TO MEMORY MANAGER CODE



```
	no_sys,		/* 12 = chdir	*/
	no_sys,		/* 13 = time	*/
	no_sys,		/* 14 = mknod	*/
	no_sys,		/* 15 = chmod	*/
	no_sys,		/* 16 = chown	*/
	do_brk,		/* 17 = break	*/
	no_sys,		/* 18 = stat	*/
	no_sys,		/* 19 = lseek	*/
	do_getset, /* 20 = getpid	*/
	no_sys,		/* 21 = mount	*/
	no_sys,		/* 22 = umount	*/
	do_getset, /* 23 = setuid	*/
	do_getset, /* 24 = getuid	*/
	no_sys,		/* 25 = stime	*/
	no_sys,		/* 26 = (ptrace)*/
	do_alarm, /* 27 = alarm	*/
	no_sys,		/* 28 = fstat	*/
	do_pause, /* 29 = pause	*/
	no_sys,		/* 30 = utime	*/
	no_sys,		/* 31 = (stty)	*/
	no_sys,		/* 32 = (gtty)	*/
	no_sys,		/* 33 = access	*/
	no_sys,		/* 34 = (nice)	*/
	no_sys,		/* 35 = (ftime)	*/
	no_sys,		/* 36 = sync	*/
	do_kill,	/* 37 = kill	*/
	no_sys,		/* 38 = unused	*/
	no_sys,		/* 39 = unused	*/
	no_sys,		/* 40 = unused	*/
	no_sys,		/* 41 = dup	*/
	no_sys,		/* 42 = pipe	*/
	no_sys,		/* 43 = times	*/
	no_sys,		/* 44 = (prof)	*/
	no_sys,		/* 45 = unused	*/
	do_getset, /* 46 = setgid	*/
	do_getset, /* 47 = getgid	*/
	do_signal, /* 48 = sig*/
	no_sys,		/* 49 = unused	*/
	no_sys,		/* 50 = unused	*/
	no_sys,		/* 51 = (acct)	*/
	no_sys,		/* 52 = (phys)	*/
	no_sys,		/* 53 = (lock)	*/
	no_sys,		/* 54 = ioctl	*/
	no_sys,		/* 55 = unused	*/
	no_sys,		/* 56 = (mpx)	*/
	no_sys,		/* 57 = unused	*/
	no_sys,		/* 58 = unused	*/
	do_exec,	/* 59 = exece	*/
	no_sys,		/* 60 = umask	*/
	no_sys,		/* 61 = chroot	*/
	no_sys,		/* 62 = unused	*/
	no_sys,		/* 63 = unused	*/

```

# Appendix B-34 - MODIFICATIONS TO MEMORY MANAGER CODE



```
            do_ksig,    /* 64 = KSIG: signals originating in the kernel        */
            no_sys,             /* 65 = UNPAUSE   */
            do_brk2,    /* 66 = BRK2 (used to tell MM size of FS,INIT) */
            no_sys,             /* 67 = REVIVE      */
            no_sys,             /* 68 = TASK_REPLY        */
#ifdef i8088
#ifdef AM_KERNEL
            do_amoeba,          /* 69 = AMOEBA SYSTEM CALL */
#else
            no_sys,             /* 69 = AMOEBA SYSTEM CALL */
#endif
#endif i8088
#ifdef SWAPER
>           do_swin,  /* 70 = swap in */
>           do_swout, /* 71 = swap out */
#else
            no_sys,             /* 70 = swap in */
            no_sys,             /* 71 = swap out */
#endif
};
```

# APPENDIX C - MODIFICATIONS TO FILE SYSTEM CODE

# Appendix C-2 - MODIFICATIONS TO FILE SYSTEM CODE



```
  /*===================================================================*
   *                          fs_init                            *
   *===================================================================*/
  PRIVATE fs_init()
  {
  /* Initialize global variables, tables, etc. */

    register struct inode *rip;
    int i;
> extern struct inode *get_inode(), *swap_node;

    buf_pool();                          /* initialize buffer pool */
    load_ram();                          /* Load RAM disk from root diskette. */
    load_super();                        /* Load super block for root device */

    /* Initialize the 'fproc' fields for process 0 and process 2. */
    for (i = 0; i < 3; i+= 2) {
            fp = &fproc[i];
            rip = get_inode(ROOT_DEV, ROOT_INODE);
            fp->fp_rootdir = rip;
            dup_inode(rip);
            fp->fp_workdir = rip;
            fp->fp_realuid = (uid) SYS_UID;
            fp->fp_effuid = (uid) SYS_UID;
            fp->fp_realgid = (gid) SYS_GID;
            fp->fp_effgid = (gid) SYS_GID;
            fp->fp_umask =  0;
    }
> swap_node = NIL_INODE;

    /* Certain relations must hold for the file system to work at all. */
    if (ZONE_NUM_SIZE != 2) panic("ZONE_NUM_SIZE != 2", NO_NUM);
    if (SUPER_SIZE > BLOCK_SIZE) panic("SUPER_SIZE > BLOCK_SIZE", NO_NUM);
    if(BLOCK_SIZE % INODE_SIZE != 0)panic("BLOCK_SIZE % INODE_SIZE != 0", NO_NUM);
    if (NR_FDS > 127) panic("NR_FDS > 127", NO_NUM);
    if (NR_BUFS < 6) panic("NR_BUFS < 6", NO_NUM);
    if (sizeof(d_inode) != 32) panic("inode size != 32", NO_NUM);
  }
```

# Appendix C-3 - MODIFICATIONS TO FILE SYSTEM CODE



```
  /* This file contains the code for performing four system calls relating to
   * status and directories.
   *
   * The entry points into this file are
   *   do_chdir:      perform the CHDIR system call
   *   do_chroot:     perform the CHROOT system call
   *   do_stat:       perform the STAT system call
   *   do_fstat:      perform the FSTAT system call
   */

  #include "../h/const.h"
  #include "../h/type.h"
  #include "../h/error.h"
  #include "../h/stat.h"
  #include "const.h"
  #include "type.h"
  #include "file.h"
  #include "fproc.h"
  #include "glo.h"
  #include "inode.h"
  #include "param.h"
  extern struct inode *swap_node;
  char swap_dir[] = {"/usr/swap"};

  /*===========================================================================*
   *                              do_chdir                                     *
   *===========================================================================*/
  PUBLIC int do_chdir()
  {
  /* Change directory.  This function is  also called by MM to simulate a chdir
   * in order to do EXEC, etc.
   */

    register struct fproc *rfp;
    int r;

    if (who == MM_PROC_NR) {
>         if(cd_flag == 2) {
>                 if(swap_node == NIL_INODE)
>                         if( (r=change(&swap_node, swap_dir, 9)) != OK) {
>                                 return(r);
>                 }
>                 dup_inode(swap_node);
>                 put_inode(fp->fp_workdir);
>                 fp->fp_workdir = swap_node;
>                 fp->fp_effuid = SUPER_USER;
>                 return(OK);
>
>         } else {
                  rfp = &fproc[slot1];
                  put_inode(fp->fp_workdir);
                  fp->fp_workdir = (cd_flag == 1 ? fp->fp_rootdir : rfp->fp_workdir);
                  dup_inode(fp->fp_workdir);
```

```
                                fp->fp_effuid = (cd_flag == 1 ? SUPER_USER : rfp->fp_effuid);
                                return(OK);
                        }
          }

  /* Perform the chdir(name) system call. */
    return change(&fp->fp_workdir, name, name_length);
  }

  /*===========================================================================*
   *                              change                                       *
   *===========================================================================*/
  PRIVATE int change(iip, name_ptr, len)
  struct inode **iip;            /* pointer to the inode pointer for the dir */
  char *name_ptr;                /* pointer to the directory name to change to */
  int len;                       /* length of the directory name string */
                                 /* if == 0, then use name_ptr directly */
  {
  /* Do the actual work for chdir() and chroot(). */

    struct inode *rip;
    register int r;
    extern struct inode *eat_path();

    /* Try to open the new directory. */
    r = 0;
>   if(cd_flag == 2)
>     do {
>        user_path[r] = name_ptr[r];
>     } while (name_ptr[r++] != 0);

    else
      if (fetch_name(name_ptr, len, M3) != OK) return(err_code);

    if ( (rip = eat_path(user_path)) == NIL_INODE) return(err_code);
    /* It must be a directory and also be searchable. */
    if ( (rip->i_mode & I_TYPE) != I_DIRECTORY)
          r = ENOTDIR;
    else
          r = forbidden(rip, X_BIT, 0);    /* check if dir is searchable */

    /* If error, return inode. */
    if (r != OK) {
          put_inode(rip);
          return(r);
    }

    /* Everything is OK.  Make the change. */
    put_inode(*iip);             /* release the old directory */
    *iip = rip;                  /* acquire the new one */
    return(OK);
  }
```

# An Implementation of Process Swapping in MINIX
# (A Message Passing Oriented Operating System)

by

Stanley George Kobylanski

B.S., Pennsylvania State University, 1972

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1989

ABSTRACT

MINIX is a small, general purpose, client-server (transaction oriented) operating system that runs on IBM-PC compatible computers. It provides a UNIX, version 7 based interface and includes many of the standard UNIX support and utility programs. It's intended use is to provide a vehicle for teaching operating system concepts. A limitation of the system is the small amount of main memory available for user processes. This paper provides a solution to that problem by describing an implementation of process swapping.